



Fighting the Branch Predictor

guillermo.polito@inria.fr

ESUG'25

Ignacio Losiggio (Mega), **Guillermo Polito**



Quick About Me: Guille

guillermo.polito@inria.fr
@guillep



- Pronounced *gife* (guichet in FR, ~ghisheh in EN?)
- **Now:** Researcher at Inria - Lille
- Pharo Contributor since ~2010
- **Keywords:** compilers, testing, test generation
- **Interests:** tooling, benchmarking, 日本語, board games, batman, concurrency, chess



If any of that interests you, come talk to me!





method name parameter
exampleWithNumber: x

pragma
<syntaxOn: #postcard>

comment
"A "complete" Pharo syntax"

local variable
| y |

boolean literals
true & false

nil literal
not & (nil isNil)

block
ifFalse: [self perform: #add: with: x].

assignment
y := thisContext

keyword message
stack size + super size.

instance variable
byteArray

integer literals
:= #[2 2r100 8r20 16rFF].

byte array
array generated at runtime

literal array
{ -42 . #(\$a #a #'I' 'm' 'a' 1.0 1.23e2 3.14s2 1) }

symbols
do: [:each |

character
| var |

block parameter
var := Transcript

global variable
show: each class name;

cascade
show: each printString].

keyword message
^ x < y

return instruction

other method definition examples:
unary
+ binaryMessageArgument
keyword: arg
keyword: arg1 withTwo: arg2

PLACE
STAMP
HERE



method name
exampleWithNumber: x
parameter

pragma
<syntaxOn: #postcard>
comment
"A "complete" Pharo syntax"

local variable
| y |
boolean literals
true & false
nil literal
nil
binary message
&
unary message
not
block
(nil isNil)
keyword message
perform: #add: with: x
assignment
y := thisContext stack size + super size.
pseudo variables
thisContext
super
instance variable
byteArray := #[2 2r100 8r20 16rFF].
integer literals
2 2r100 8r20 16rFF
byte array
array generated at runtime
{ -42 . #(\$a #a #'I' 'm' 'a' 1.0 1.23e2 3.14s2 1) }
literal array
do: [:each |
symbols
character
string
floating point
scaled decimal
local block variable
| var |
block parameter
global variable
var := Transcript
show: each class name
show: each printString].
keyword message
show: each printString
keyword message
^ x < y
return instruction

other method definition examples:
unary
+ binaryMessageArgument
keyword: arg
keyword: arg1 withTwo: arg2

PLACE
STAMP
HERE

User defined operations

.....

.....

.....

.....

.....



method name
exampleWithNumber: x
parameter

pragma
<syntaxOn: #postcard>
comment
"A "complete" Pharo syntax"

local variable
| y |
boolean literals
true & false
binary message
not &
nil literal
(nil isNil)
unary message
block
keyword message
ifFalse: [self perform: #add: with: x].
assignment
pseudo variables
y := thisContext stack size + super size.
instance variable
integer literals
byteArray := #[2 2r100 8r20 16rFF].
array generated at runtime
literal array
{ -42 . #(\$a #a #'I' 'm' 'a' 1.0 1.23e2 3.14s2 1) }
symbols
character
string
floating point
scaled decimal
do: [:each |
local block variable
block parameter
| var |
global variable
var := Transcript
show: each class name;
show: each printString].
keyword message
keyword message
^ x < y
return instruction

other method definition examples:
unary
+ binaryMessageArgument
keyword: arg
keyword: arg1 withTwo: arg2

PLACE
STAMP
HERE

User defined operations

Control Flow



```

exampleWithNumber: x

<syntaxOn: #postcard>
"A "complete" Pharo syntax"

| y |
true & false not & (nil isNil)
ifFalse: [ self perform: #add: with: x ].
y := thisContext stack size + super size
byteArray := #[2 2r100 8r20 16rFF].

{ -42 . #($a #a #'I''n' 'a' 1.0 1.23e2 3.14s2 1) }
do: [ :each |
    | var |
    var := Transcript
    show: each class name;
    show: each printString ].

^ x < y

```

PLACE
STAMP
HERE

User defined operations

Control Flow

Operators

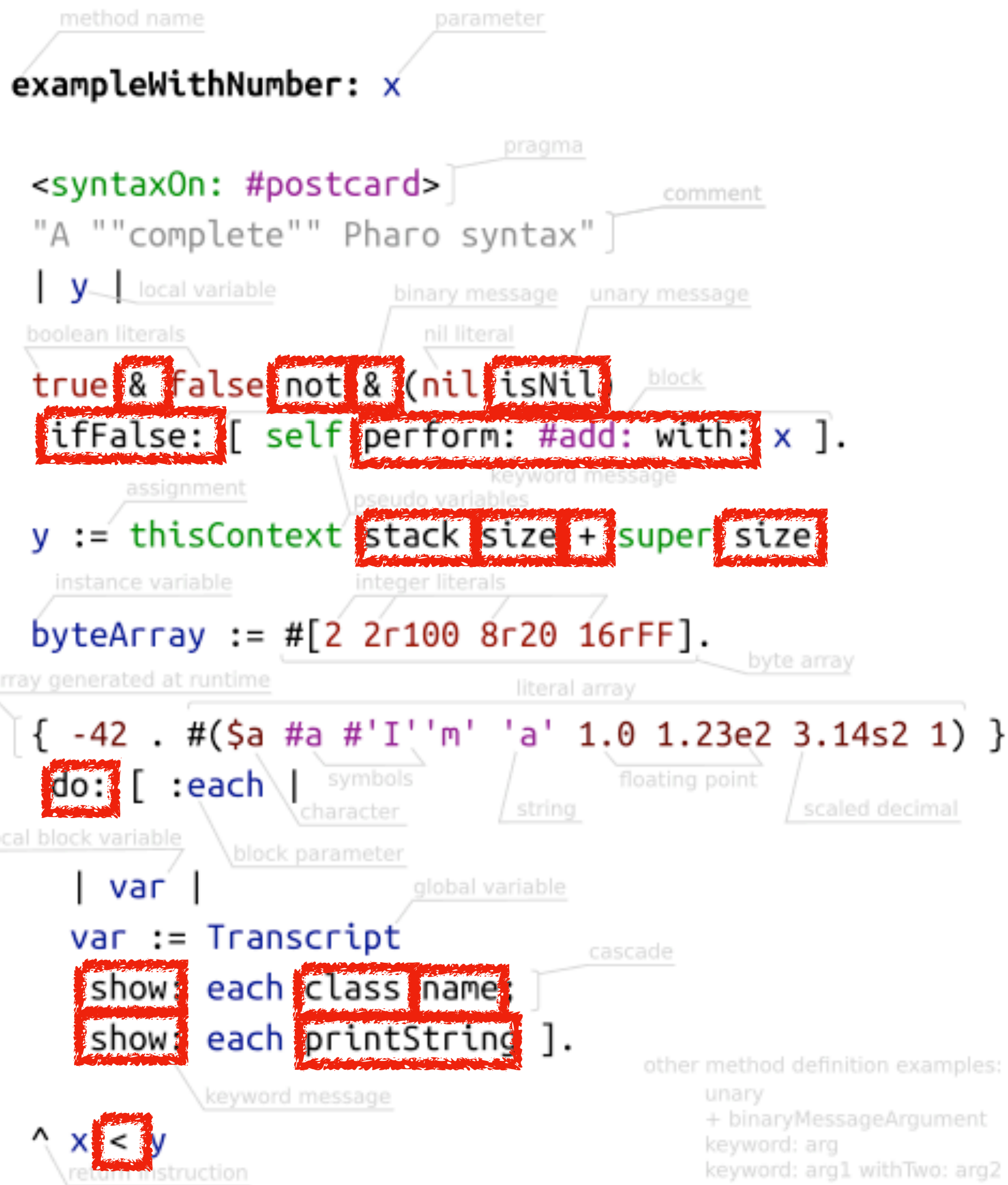


User defined operations

Control Flow

Operators

Reflection



PLACE
STAMP
HERE

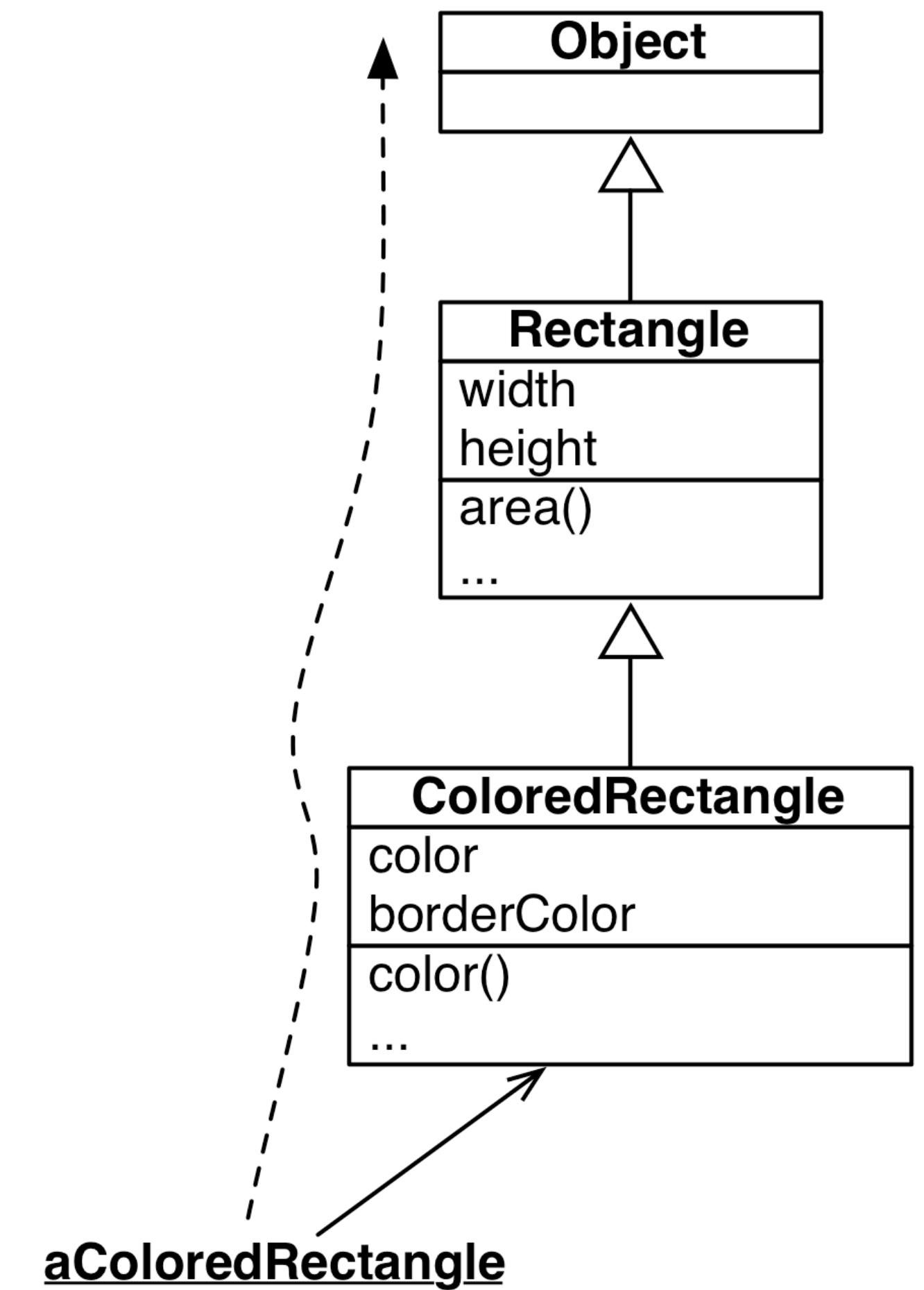
User defined operations

Control Flow
Messages
Operators

Reflection

~~an Object~~ Everything is a Message

aColoredRectangle width



~~an Object~~ Everything is a Message

What you see:

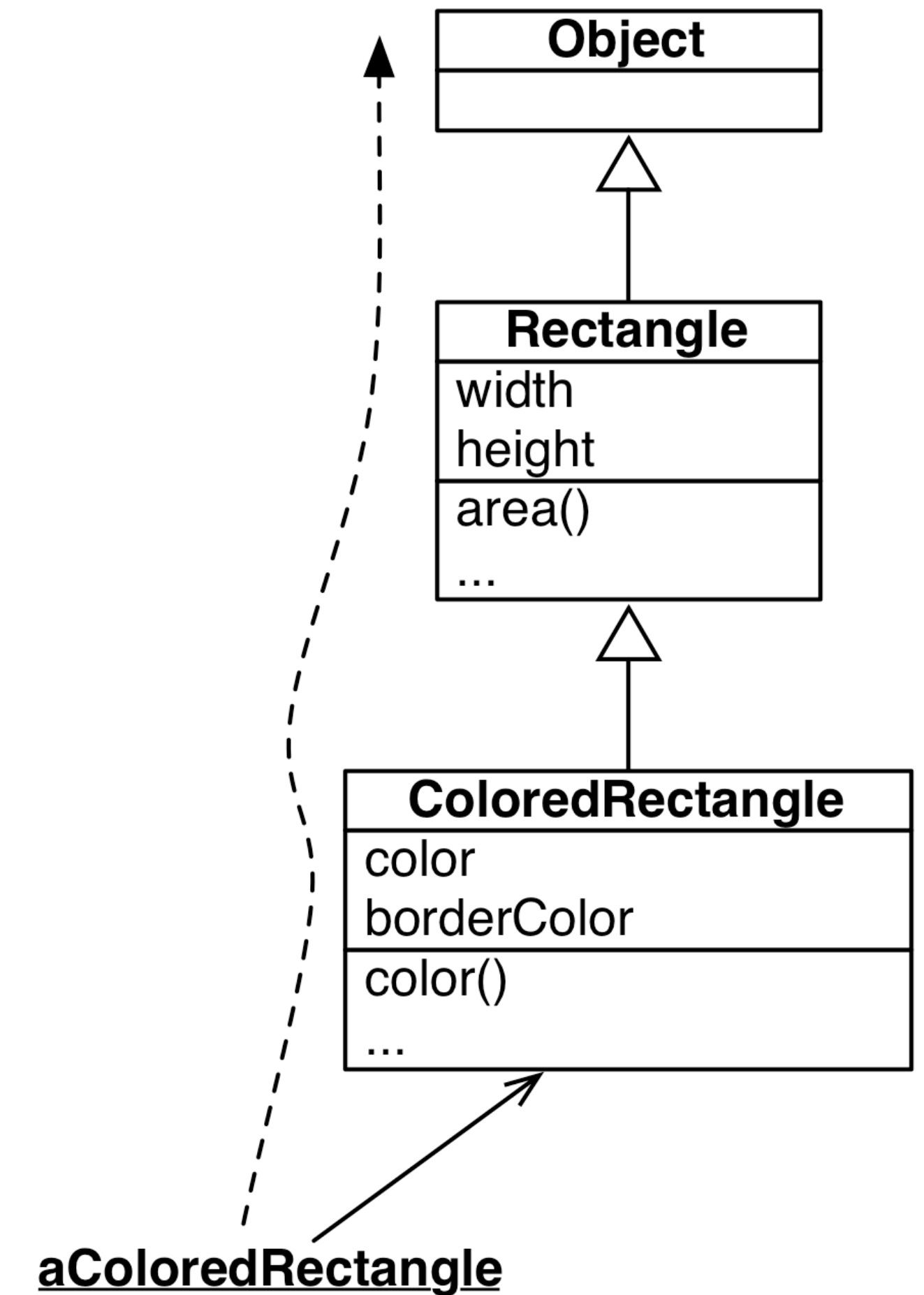
`aColoredRectangle width`

What you get:

`runtime`

`apply: (runtime lookup: #width in: aColoredRectangle))`

`on: aColoredRectangle`



~~an Object~~ Everything is a Message

What you see:

`aColoredRectangle` width

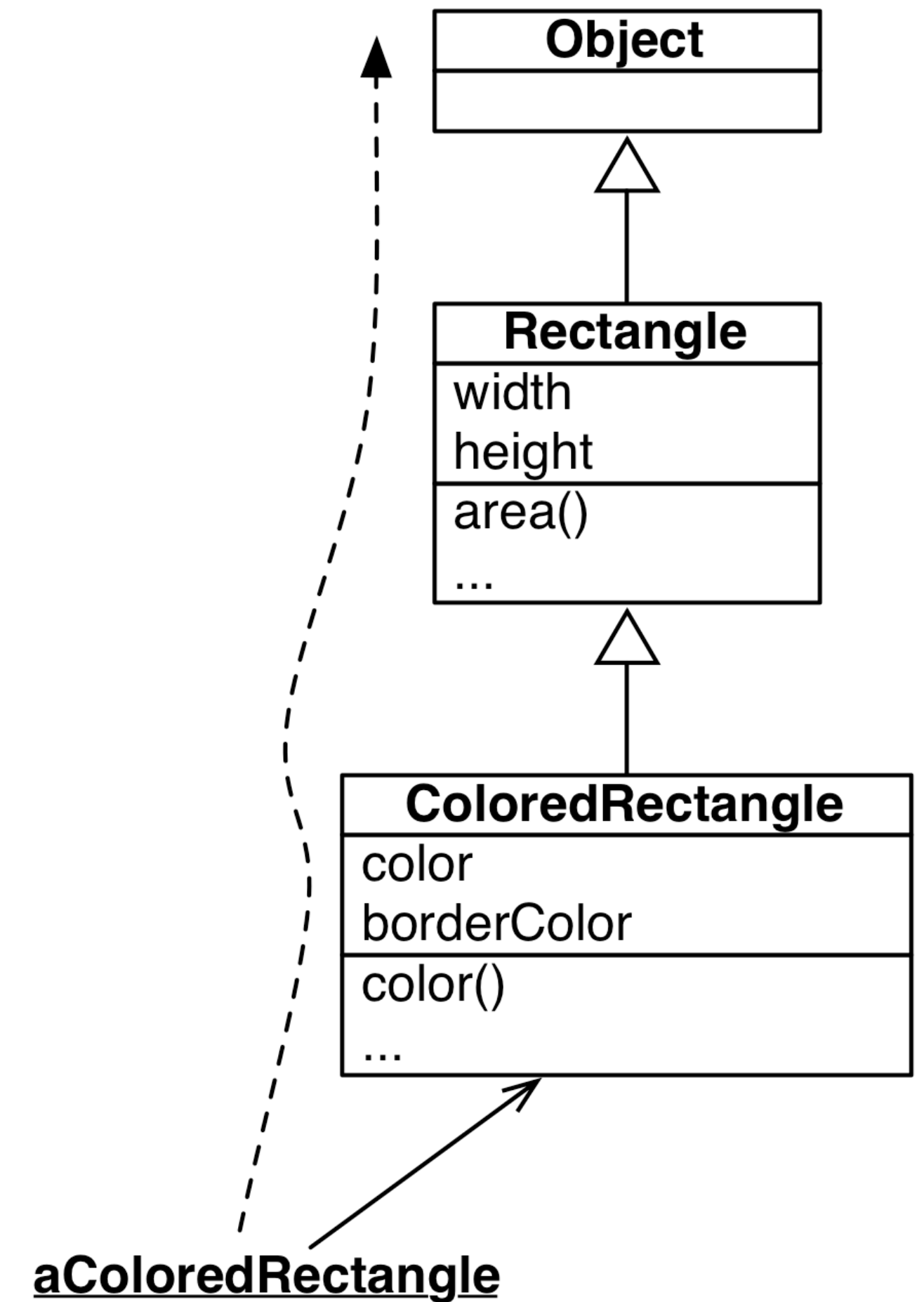
What you get:



`runtime`

`apply: (runtime lookup: #width in: aColoredRectangle))`

`on: aColoredRectangle`



Dynamic Translation - JIT Compilation

- Do not improve instruction dispatch and lookup: **remove** them !
- Translate functions/methods to machine code on the fly
- Techniques: dynamic compilation, code copying...

Efficient Implementation of the Smalltalk-80 System

L. Peter Deutsch

Xerox PARC, Software Concepts Group

Allan M. Schiffman

Fairchild Laboratory for Artificial Intelligence Research

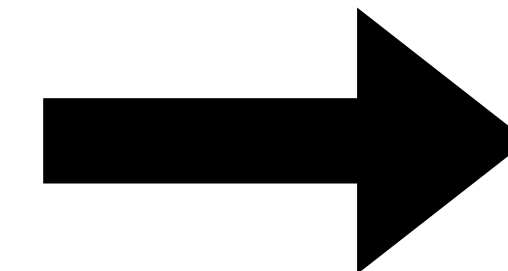
ABSTRACT

programming language includes dynamic upward funargs, and universally the Smalltalk-80 programming system on with incremental compilation, and ty. These features of modern among the most difficult to implement ally. A new implementation of the

machine instruction set, similar to the Pascal P-system [Ammann 75] [Ammann 77]. One unusual feature of the Smalltalk-80 v-machine is that it makes runtime state such as procedure activations visible to the programmer as data objects. This is similar to the "spaghetti stack" model of Interlisp [XSIS 83], but more straightforward: Interlisp uses a programmer-visible indirection mechanism to reference procedure activations, whereas the Smalltalk-80 programmer treats procedure activations just like any other data objects.

```
void interpret(){
  while (1){
    switch(nextInstruction){
      case push: ...
      case pop: ...
      case send: ...
      ...
    }
  }
}
```

12



```
void m1(){
  push;
  send;
  push;
  send;
  pop;
}
```

```
void m2(){
  push;
  send;
  send;
  pop;
}
```

```
void m3(){
  push;
  send;
  pop;
}
```

Message Sends in the Cogit JIT Compiler

`aColoredRectangle` width

```
ldr x23, [x28]
```

```
ldr x22, #12
```

```
bl #-248
```


Message Sends in the Cogit JIT Compiler

`aColoredRectangle` width

LOAD x23, [x28]

LOAD x22, #12

CALL #-248

Message Sends in the Cogit JIT Compiler

aColoredRectangle width

LOAD aColoredRectangle

LOAD x22, #12

CALL #-248

Message Sends in the Cogit JIT Compiler

aColoredRectangle width

LOAD aColoredRectangle

LOAD #width

CALL #-248

Message Sends in the Cogit JIT Compiler

aColoredRectangle width

LOAD aColoredRectangle

LOAD #width

CALL theLookupRoutine

Unlinked

aColoredRectangle width

LOAD aColoredRectangle

LOAD #width

CALL theLookupRoutine

→ method := self lookup

method invokeWith: receiver

Inline Caching

aColoredRectangle width

LOAD aColoredRectangle

LOAD #width

CALL theLookupRoutine.....

→ method := self lookup

self cache: method

method invokeWith: receiver

Efficient Implementation of the Smalltalk-80 System

L. Peter Deutsch
Xerox PARC, Software Concepts Group

Allan M. Schiffman
Fairchild Laboratory for Artificial Intelligence Research

Linking the Send Site

aColoredRectangle width

LOAD aColoredRectangle

LOAD ~~#width~~
ColoredRectangle

CALL ColoredRectangle >> #width
~~theLookupRoutine~~

method := self lookup

self cache: method

method invokeWith: receiver

Linking the Send Site

aColoredRectangle width

method := self lookup

LOAD aColoredRectangle

self cache: method

LOAD ^{#width} ColoredRectangle

method invokeWith: receiver

CALL ColoredRectangle >> ^{#width}
_{theLookupRoutine}

If this receiver's class, this is the method to call

Speculative Linkage

aColoredRectangle width

LOAD aColoredRectangle

LOAD ColoredRectangle

CALL ColoredRectangle >> #width

ColoredRectangle >> #width

Check receiver type

If wrong,

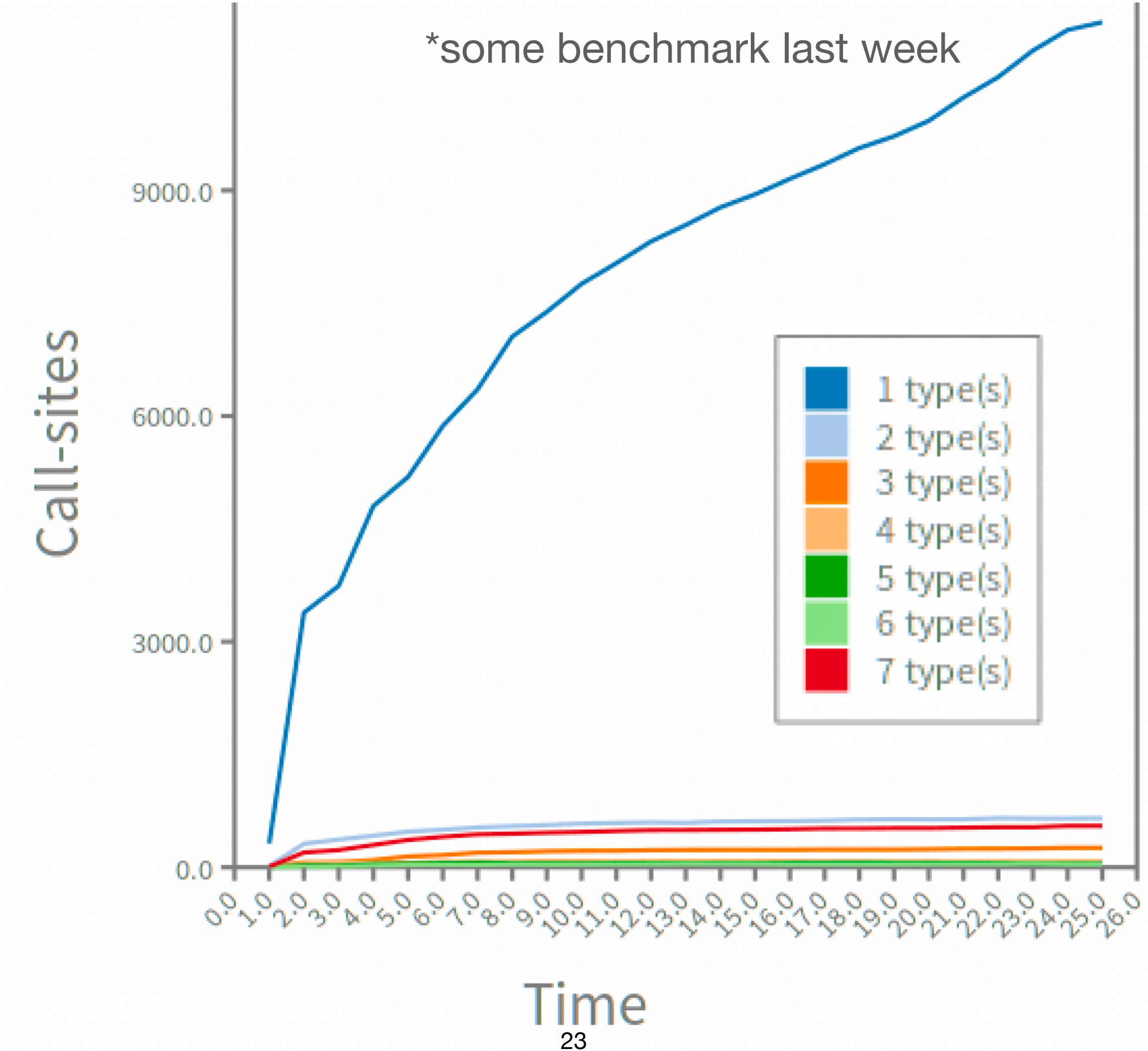
CALL theLookupRoutine

...

“method body”

...

Polymorphic Sites: fail+lookup+rewrite



Polymorphic Inline Caches

Published in ECOOP '91 proceedings, Springer Verlag Lecture Notes in Computer Science 512, July, 1991.

Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches

Urs Hölzle
Craig Chambers
David Ungar[†]

Computer Systems Laboratory, Stanford University, Stanford, CA 94305
{urs,craig,ungar}@self.stanford.edu

Abstract: *Polymorphic inline caches* (PICs) provide a new way to reduce the overhead of polymorphic message sends by extending inline caches to include more than one cached lookup result per call site. For a set of typical object-oriented SELF programs, PICs achieve a median speedup of 11%.

Caching the Last N Results

`aColoredRectangle` width

LOAD `aColoredRectangle`

LOAD `ColoredRectangle`

CALL `DispatchTable`
~~`ColoredRectangle` >> `#width`~~

Check receiver type

If ok

CALL `ColoredRectangle` >> `#width`

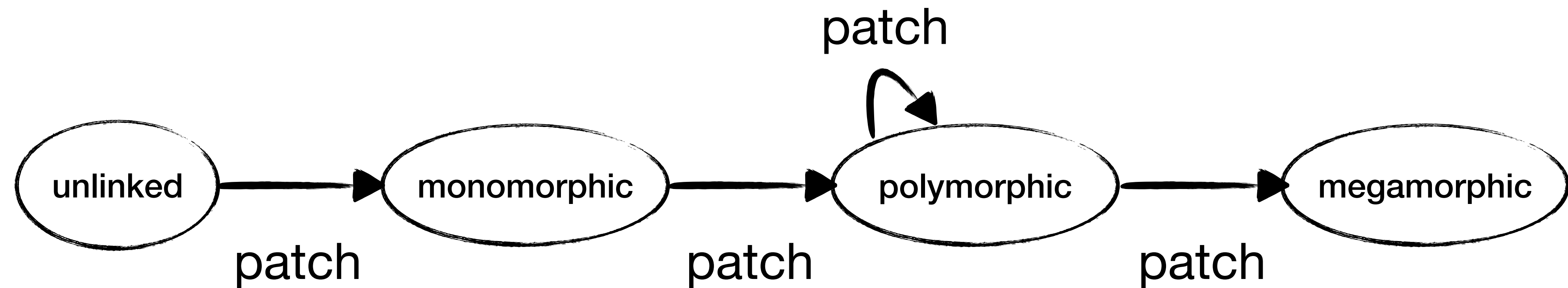
If receiver type == `Rectangle`

CALL `Rectangle` >> `#width`

...

CALL `theLookupRoutine`

It's Just a Code Patching State Machine



Patching Machine Code

- The compiler has to output ***patchable code***
- We need different patching routines **per platform**
- Some decisions are taken by ***disassembling*** the machine code at ***runtime***

Is this process suspended at a send?

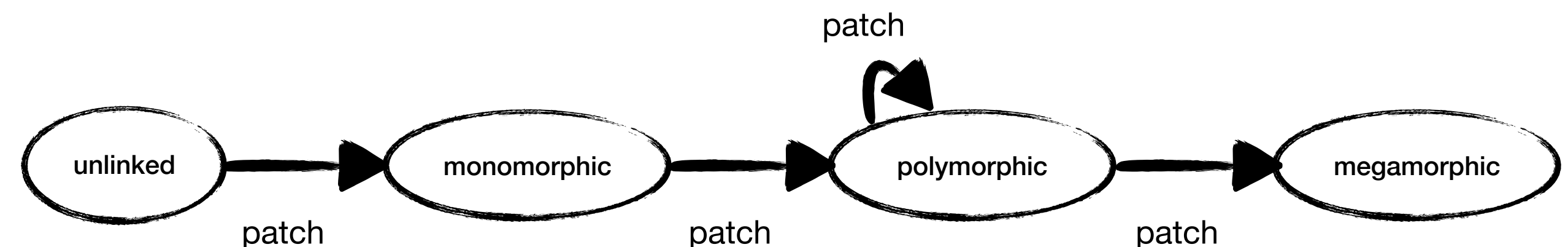
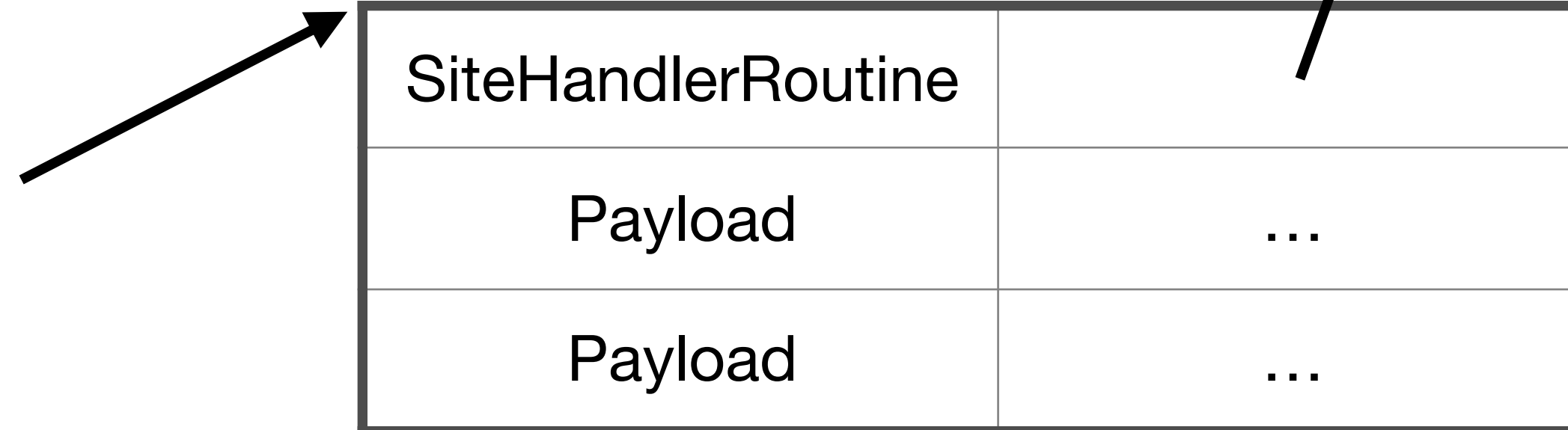


Table Based PICs



Table Based PLCs

LOAD aColoredRectangle
LOAD SiteHandlerRoutine
CALL SiteHandlerRoutine



SiteHandlerRoutine	
Payload	...
Payload	...

Some Machine Code

Patching

PICs

Tables

unlinked

```
LOAD aColoredRectangle
LOAD #width
CALL theLookupRoutine
```

SiteHandlerRoutine	theLookupRoutine
...	...
...	...

mono

```
LOAD aColoredRectangle
LOAD ColoredRectangle
CALL ColoredRectangle >> #width
```

SiteHandlerRoutine	ColoredRectangle >> #width
...	...
...	...

```
LOAD aColoredRectangle
LOAD SiteHandlerRoutine
CALL SiteHandlerRoutine
```

poly

```
LOAD aColoredRectangle
LOAD ColoredRectangle
CALL DispatchTable
```

SiteHandlerRoutine	DispatchTable
Type1	method1
Type2	method2



UBench: a Micro Benchmark Infrastructure

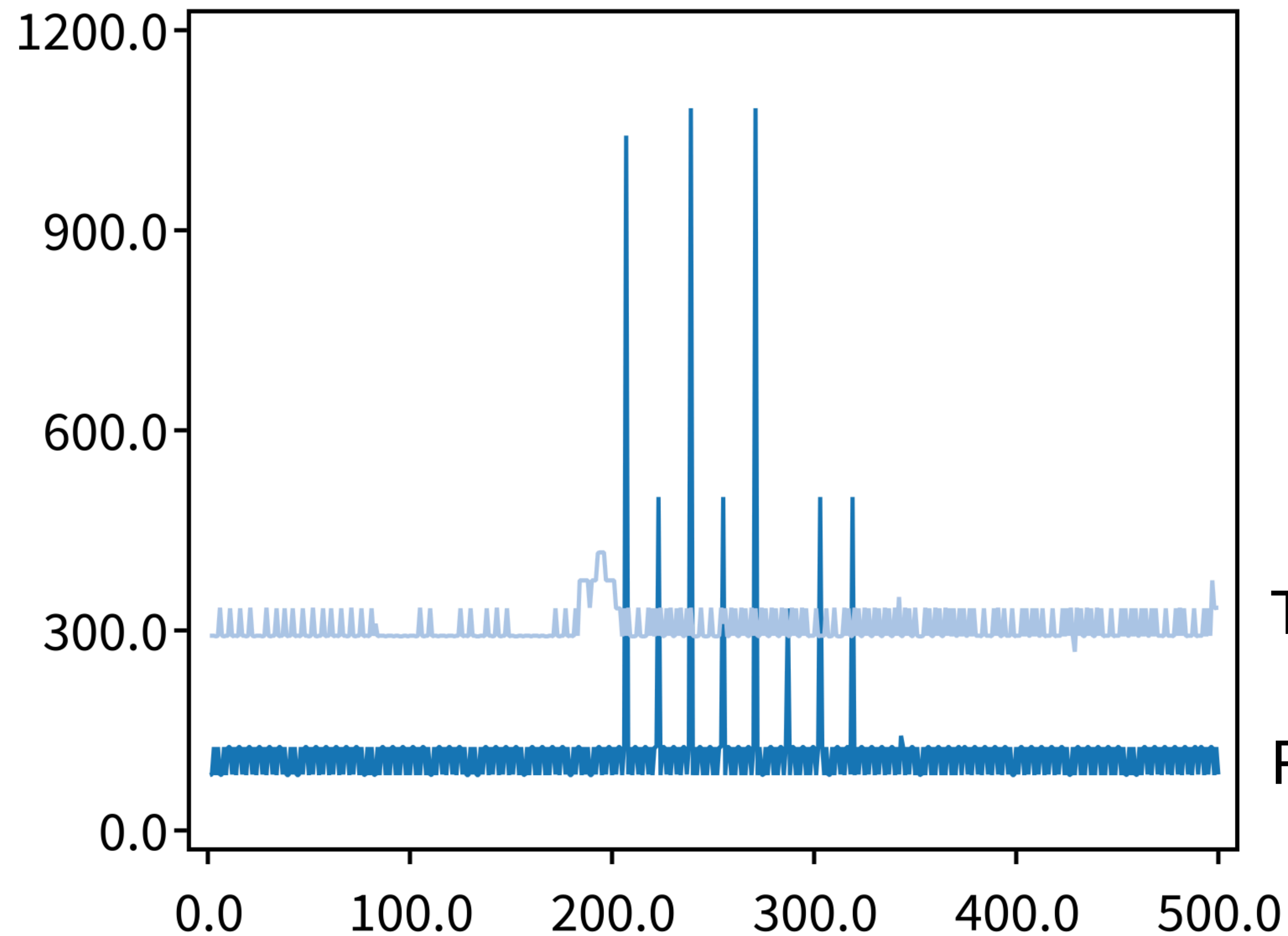
- No noise
- No GC
- No JIT compilations
- No context switches

```
self assert: jitcomps = newjitcomps.

contextSwitches := vm parameterAt: 56.
"All the loops up to 0 are warmup so we can link #sendYourselfTo: on this method"
-3 to: aReceiverSequence size do: [ :i |
    | receiver startTime endTime |
    newgcs := vm incrementalGCCCount + vm fullGCCCount.
    self assert: gcs = newgcs.
    newjitcomps := vm compiledMethodsCount + vm compiledBlocksCount.
    self assert: jitcomps = newjitcomps.
    receiver := nil.
    0 < i ifTrue: [ receiver := aReceiverSequence at: i ].
    startTime := Smalltalk highResClock.
    self sendYourselfTo: receiver.
    endTime := Smalltalk highResClock.
    0 < i ifTrue: [ result at: i put: endTime - startTime. ] ].

contextSwitchesEnd := vm parameterAt: 56.
self assert: contextSwitches = contextSwitchesEnd.
```


Wait WAT?



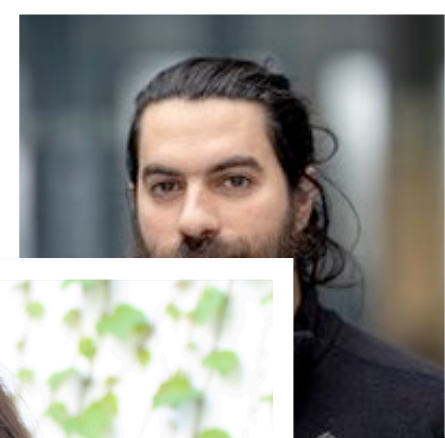
Tables: 305 +- 24 **(3x)**



Patching: 116.284 +- 85

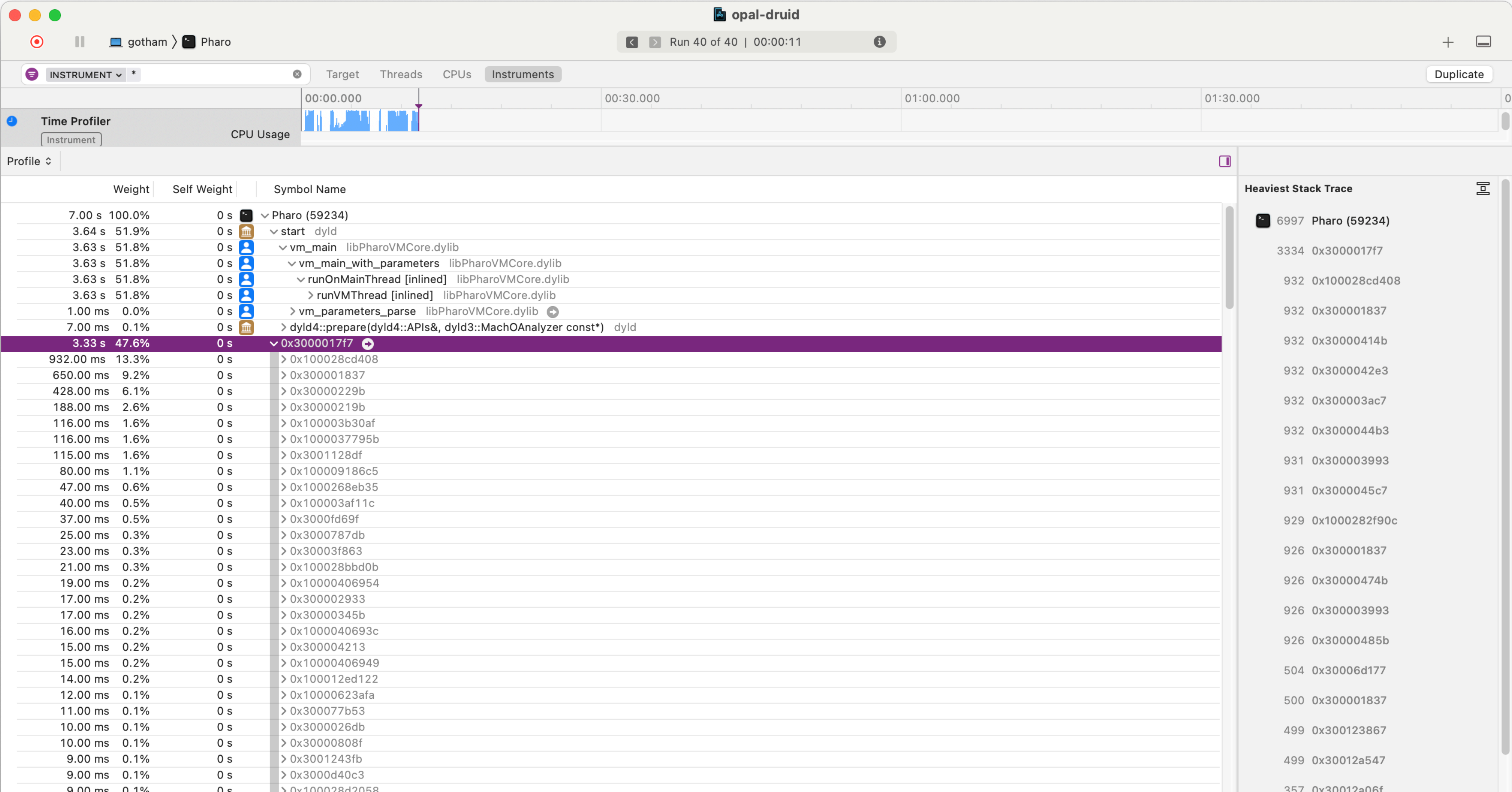


Mega (Ignacio Losiggio)



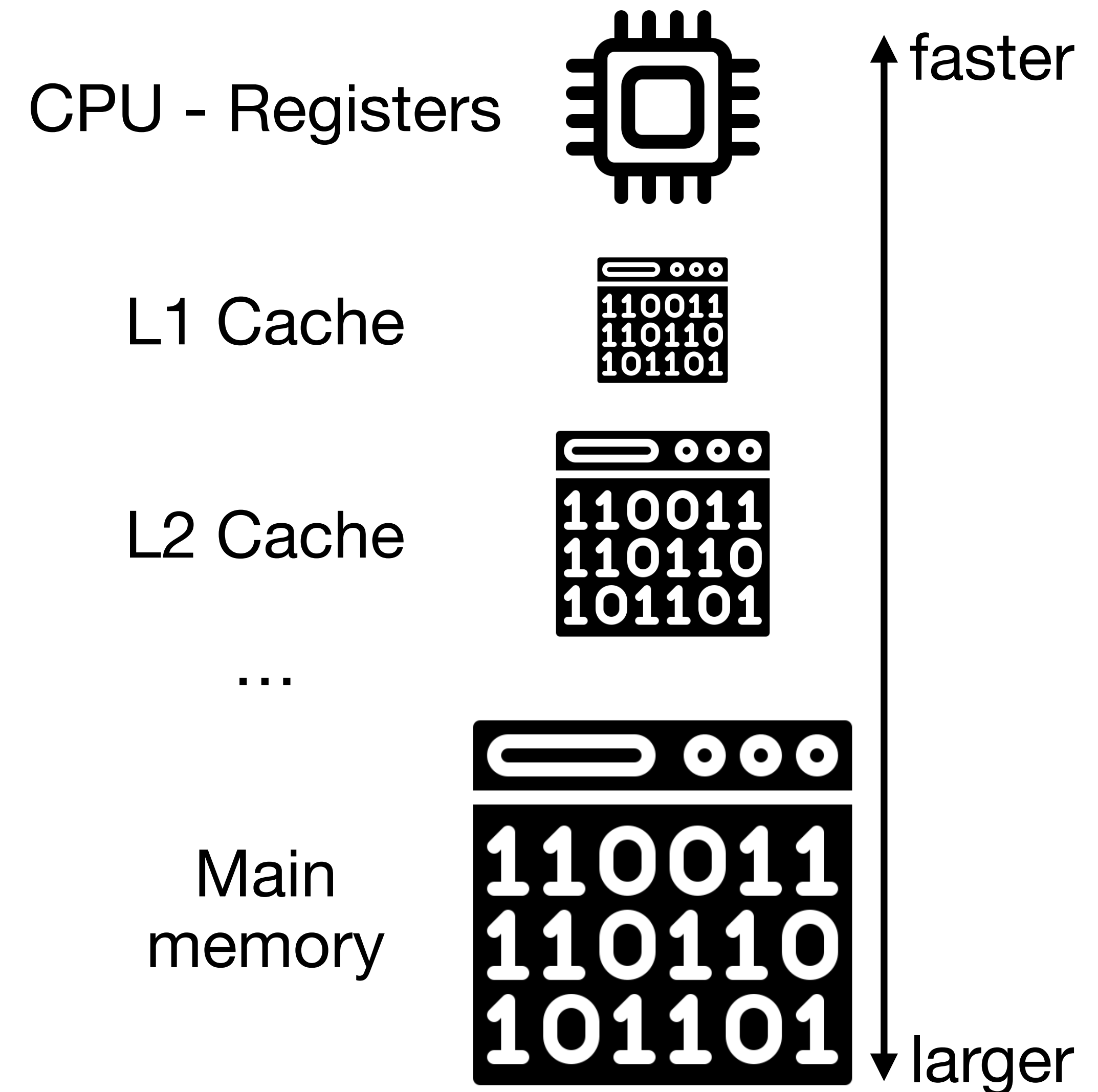
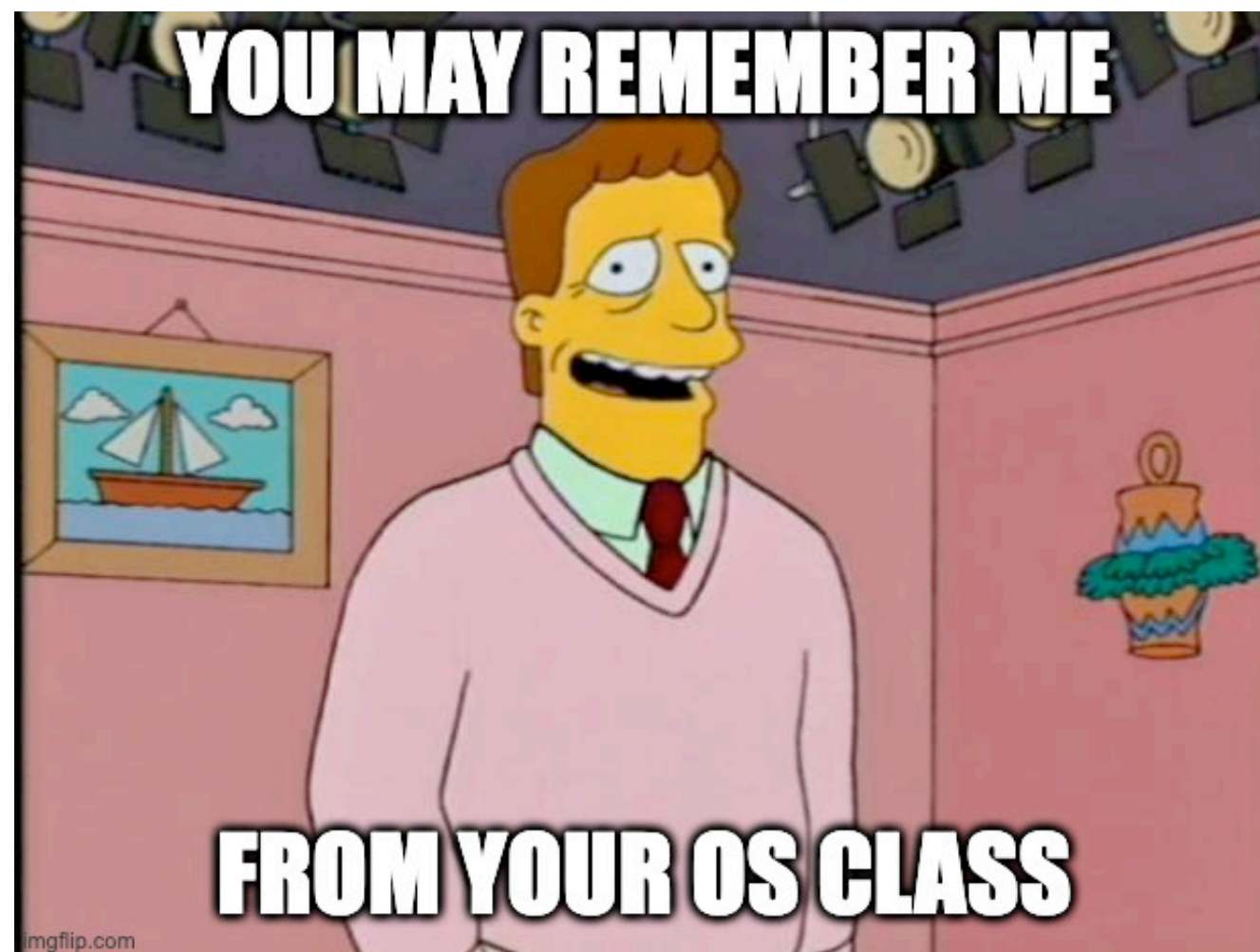
Are table PLCs
CPU unfriendly?

Where does the time go?



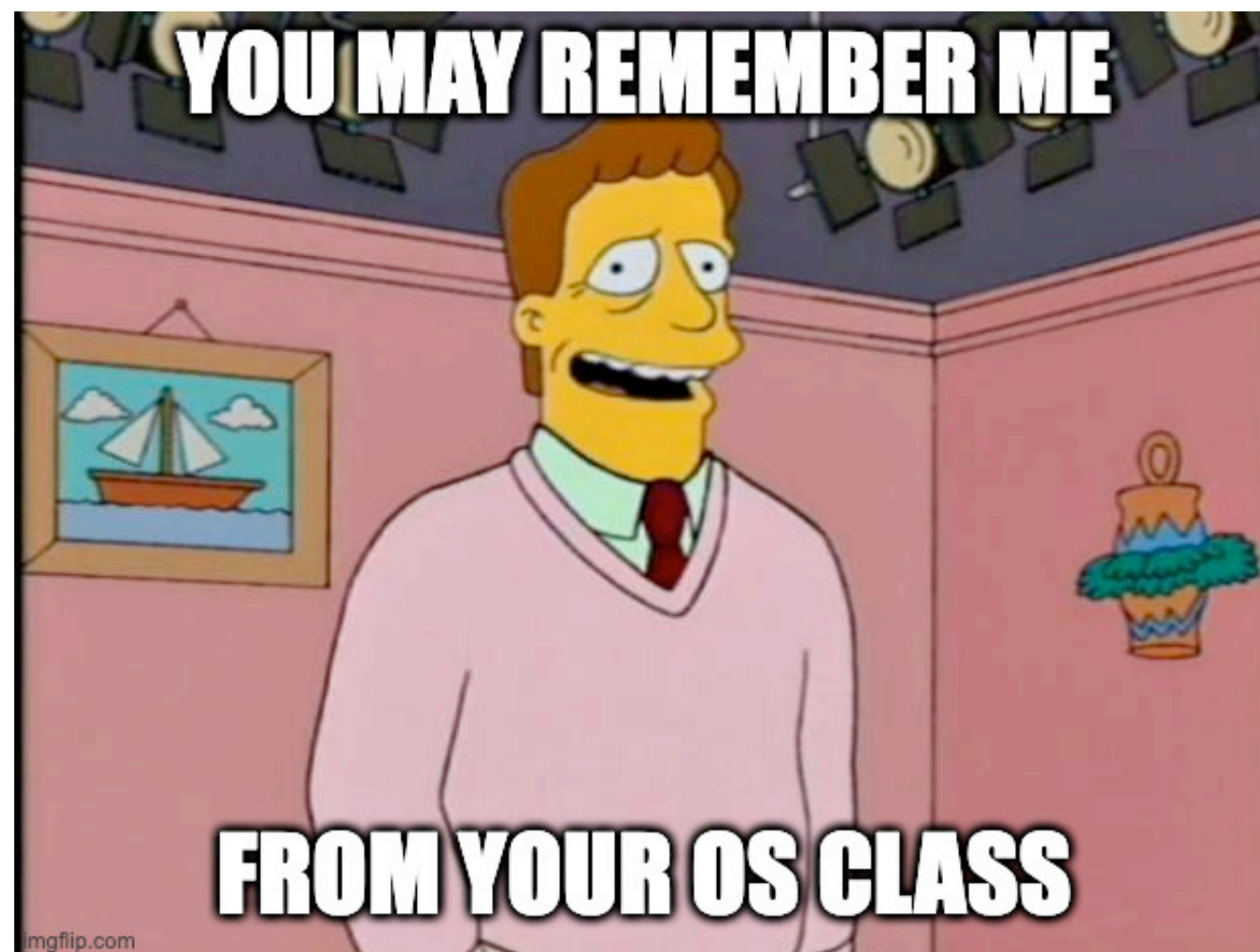
Memory Hierarchy and CPU Caches

- *Faster* when closer to CPU
- Invalid data needs to be *refetched*
- Data = program data + **instructions**

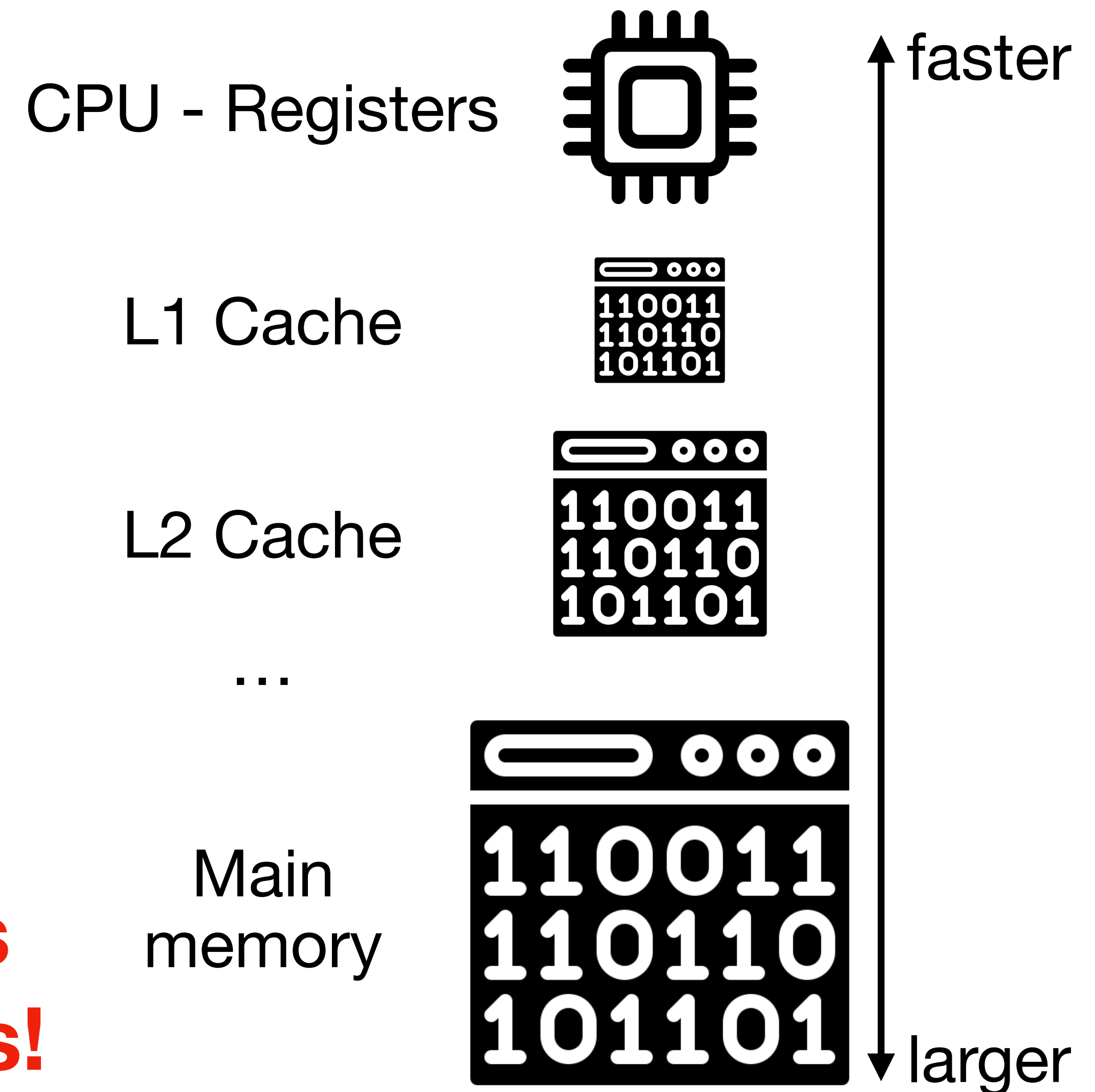


Memory Hierarchy and CPU Caches

- *Faster* when closer to CPU
- Invalid data needs to be *refetched*
- Data = program data + **instructions**



**Patching
Invalidates
Instructions!**

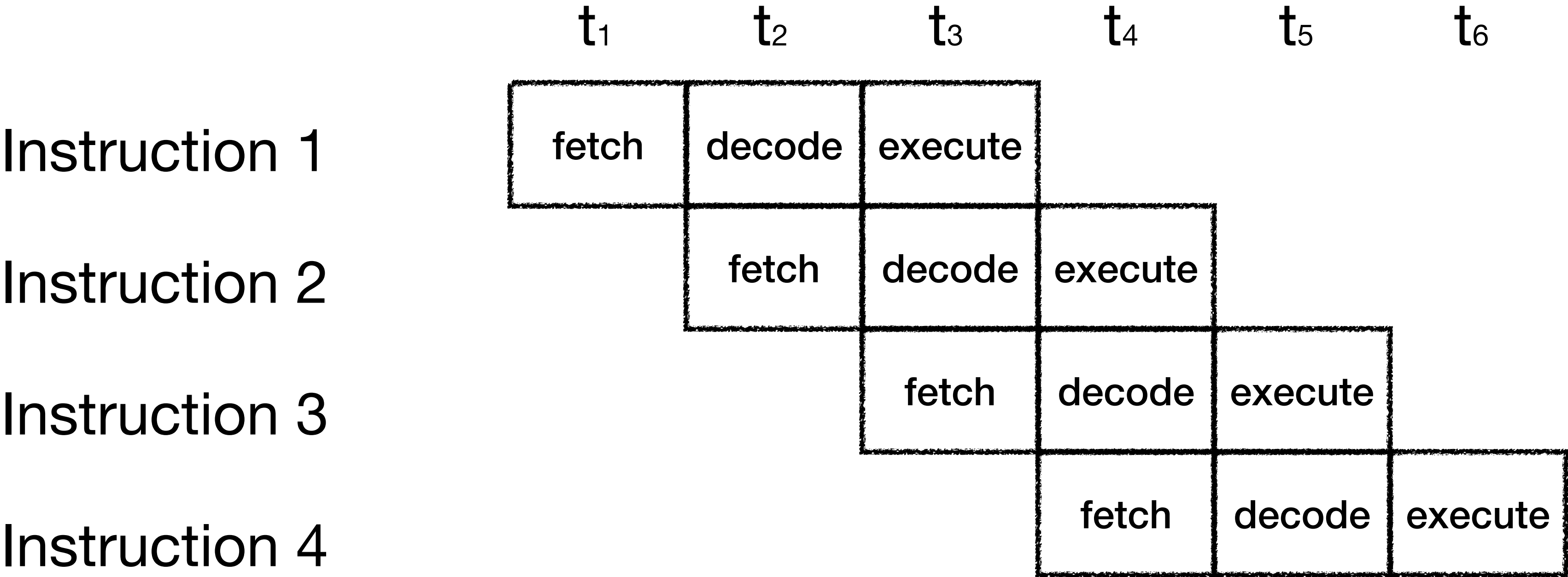


Profiling CPU Metrics: Cache Misses

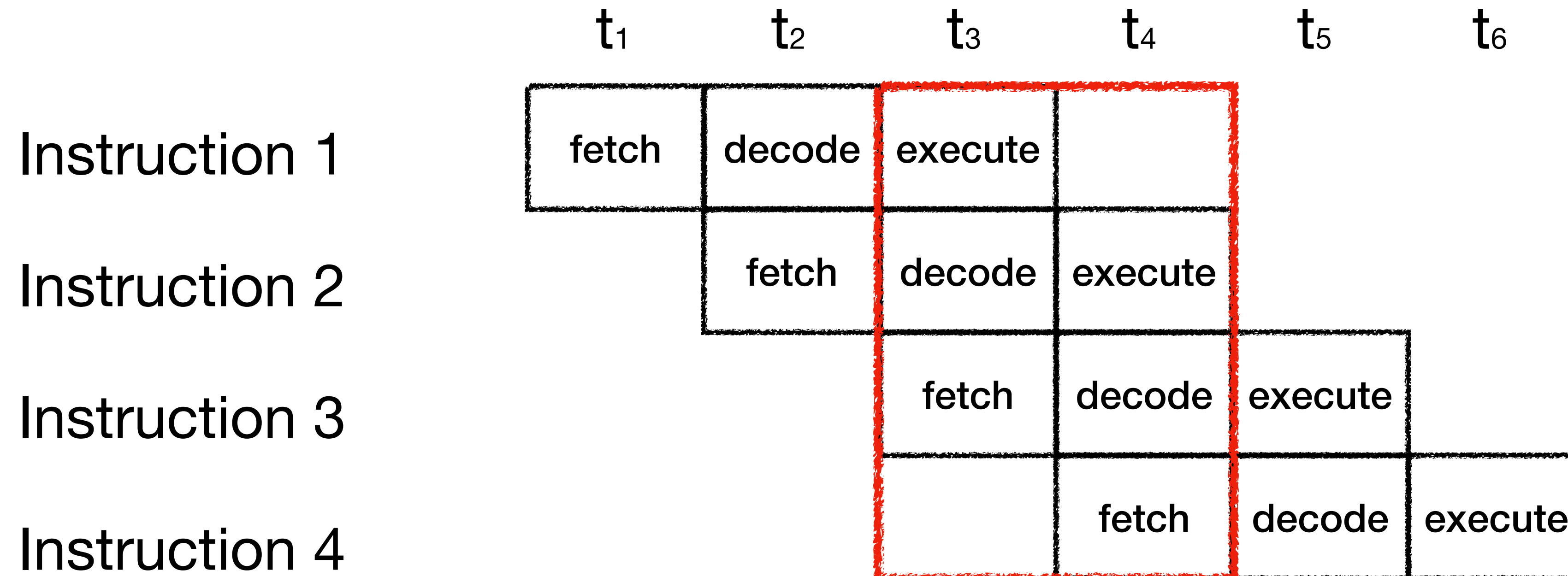
	L1D TLB_MISS	L1D MISS_LD	L1D MISS_ST	L1D WRITEBACK	L1I TLB_MISS	L2 TLB_MISS
Patching	23.9M	131.6M	70M	80M	42.6M	5.8M
Tables	9.6M	80M	25M	31M	10.7M	1M

Patching is slower here? 🤔

CPU Pipelines

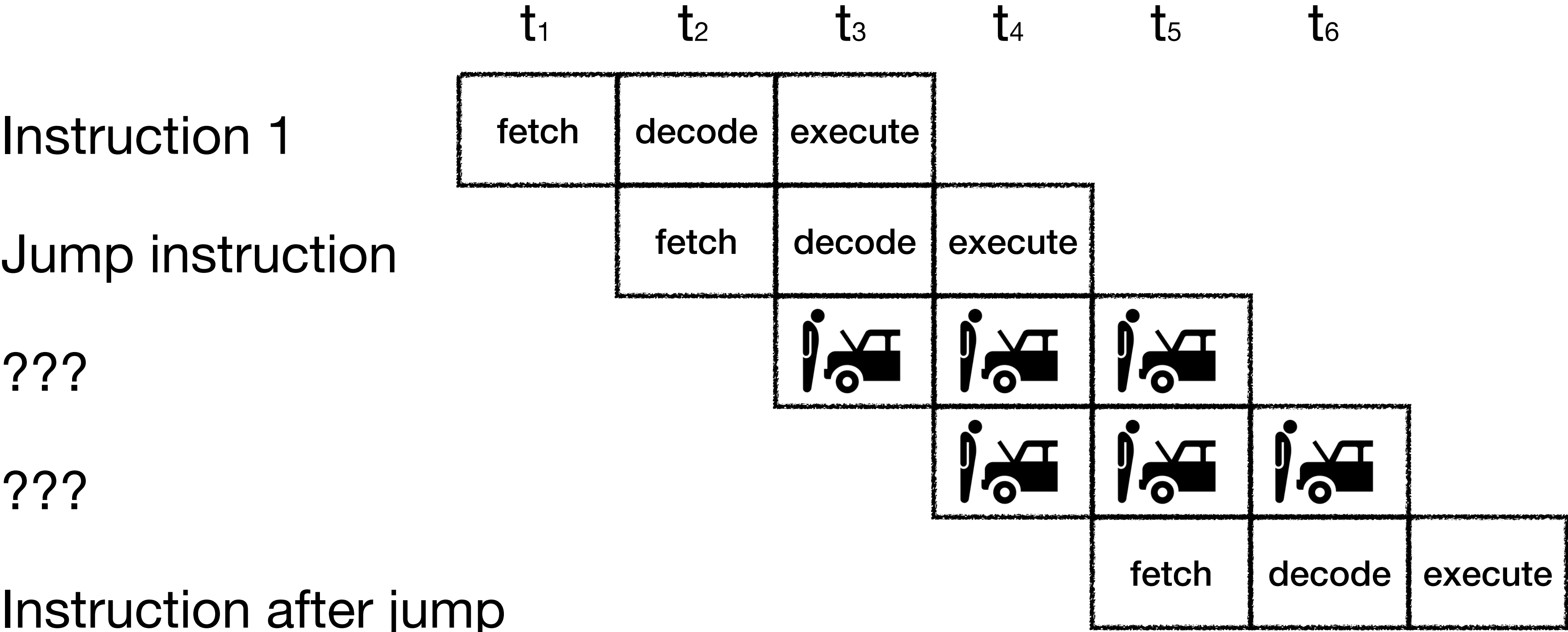


CPU Pipelines

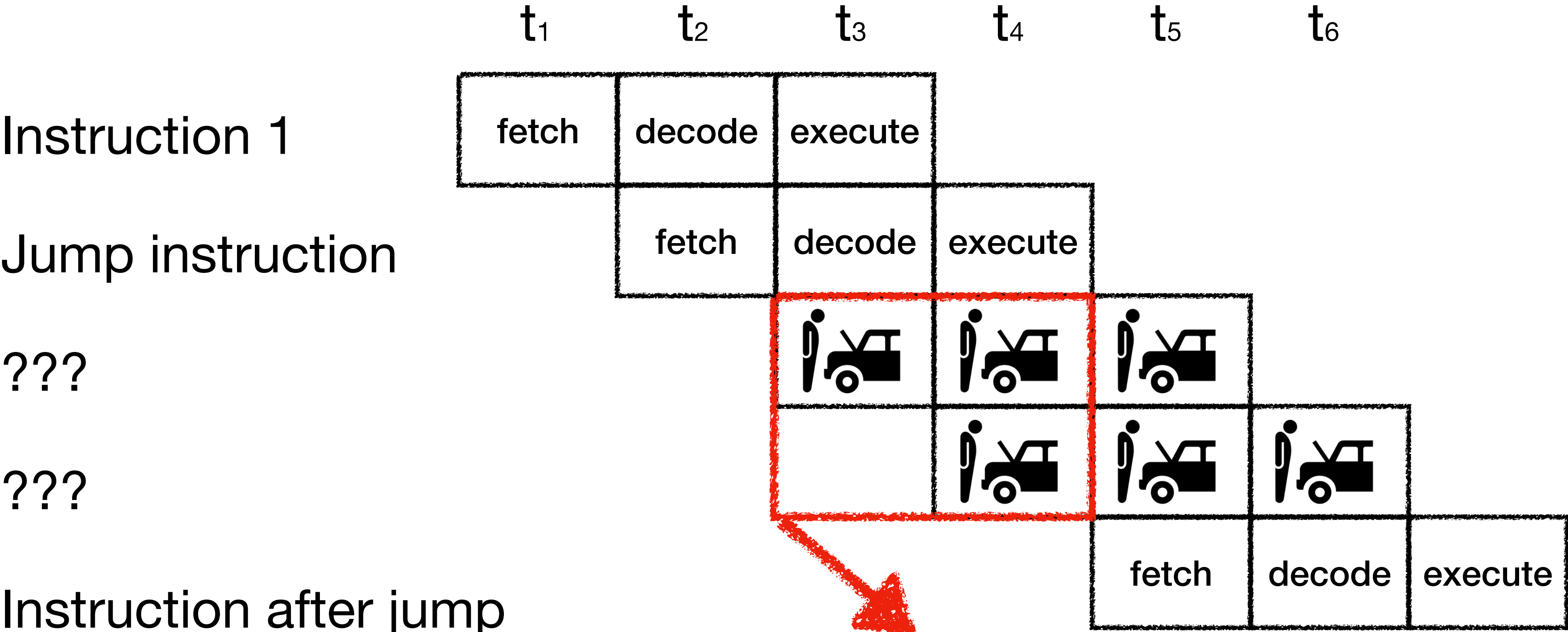


CPU Parallel Execution

CPU Pipelines and Branches

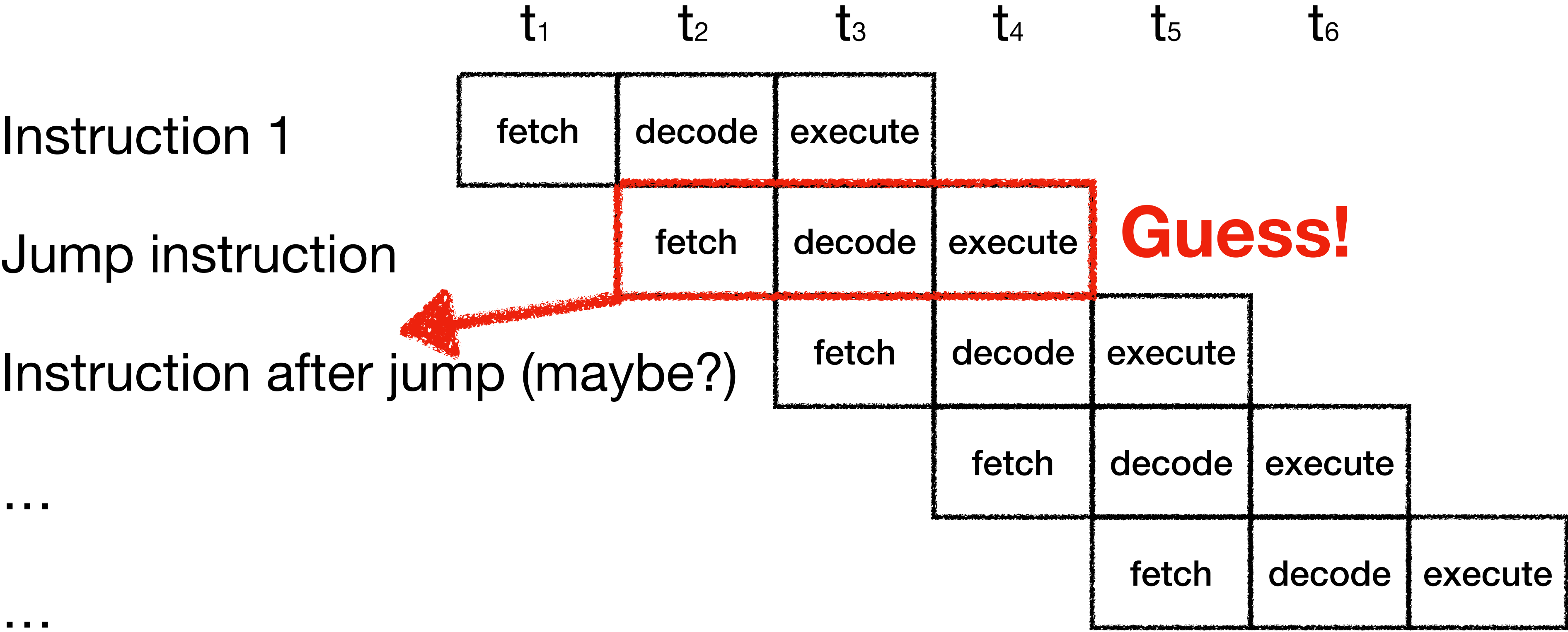


CPU Pipelines and Branches

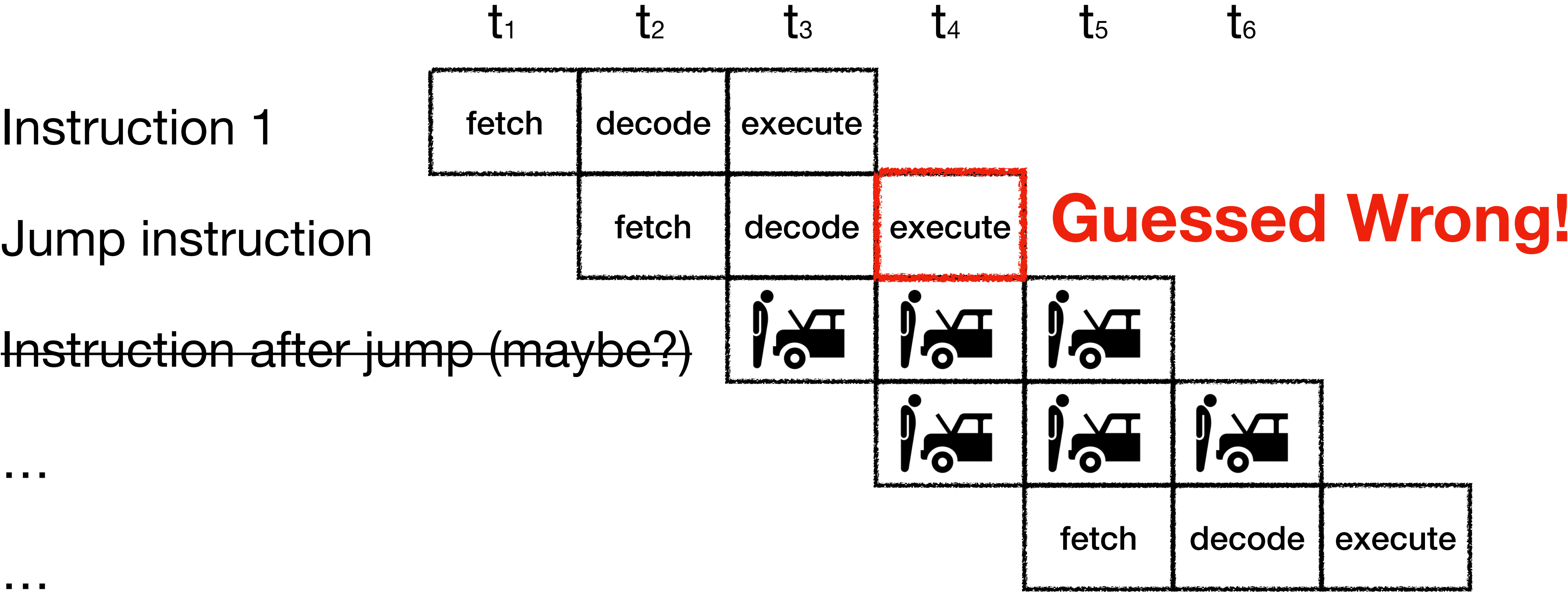


Stalling loses time!

Branch Prediction



Branch (Mis)prediction



Profiling CPU Metrics: Cache Misses

	CALL_INDIR MISPRED	BR MISPRED	BR_INDIR MISPRED	STALL
Patching	7.7K	4.6M	20M	412M
Tables	17.4K (~2.5x)	6.4M (~1.3x)	44M (~2.2x)	840M (~2x)

Well... this is better! (?) 🤔

Direct Branches and Prediction

unlinked

LOAD aColoredRectangle

LOAD #width

CALL theLookupRoutine

mono

LOAD aColoredRectangle

LOAD ColoredRectangle

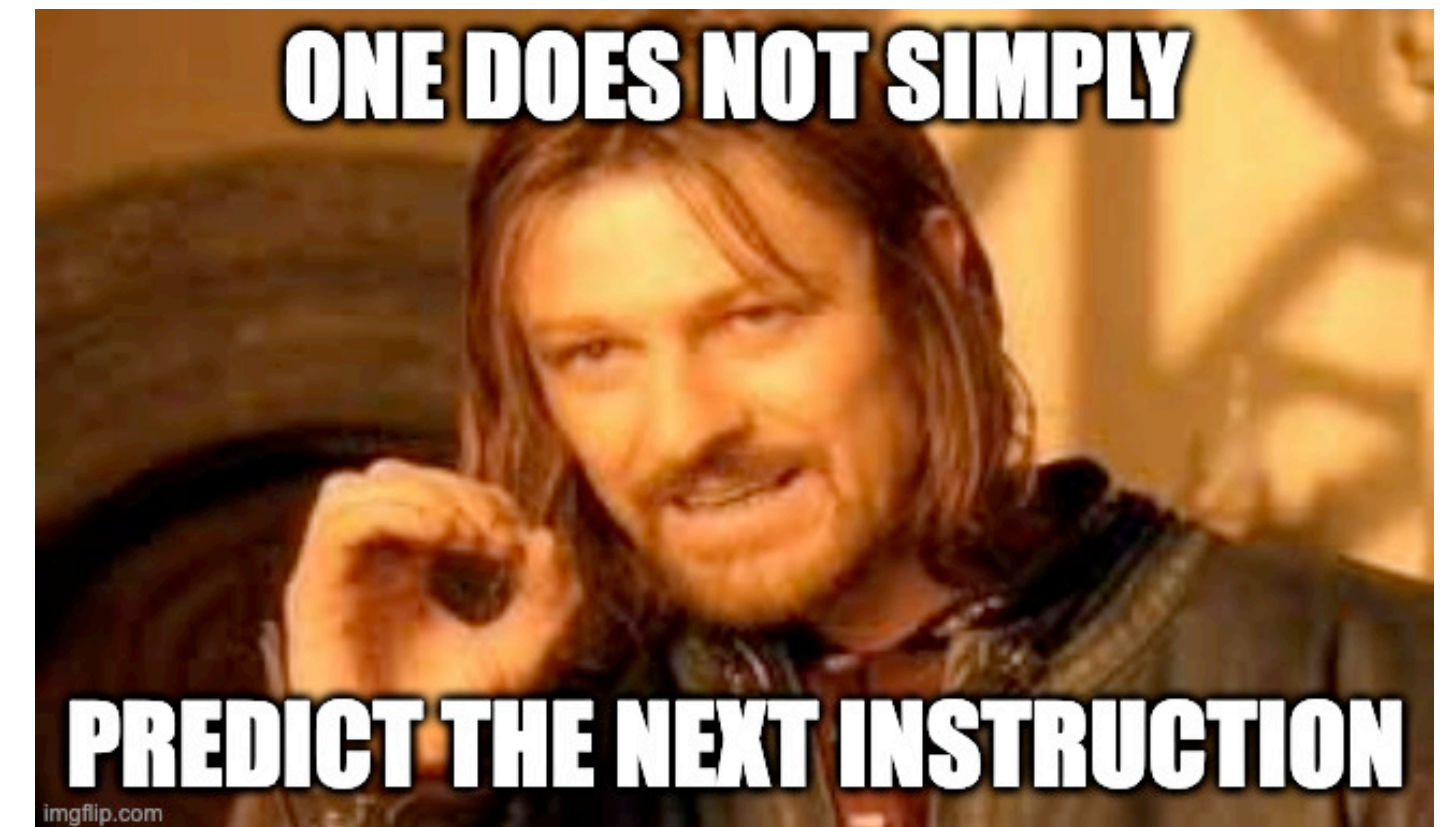
CALL ColoredRectangle >> #width

poly

LOAD aColoredRectangle

LOAD ColoredRectangle

CALL DispatchTable



Direct Branches and Prediction

unlinked

LOAD aColoredRectangle

LOAD #width

CALL theLookupRoutine

mono

LOAD aColoredRectangle

LOAD ColoredRectangle

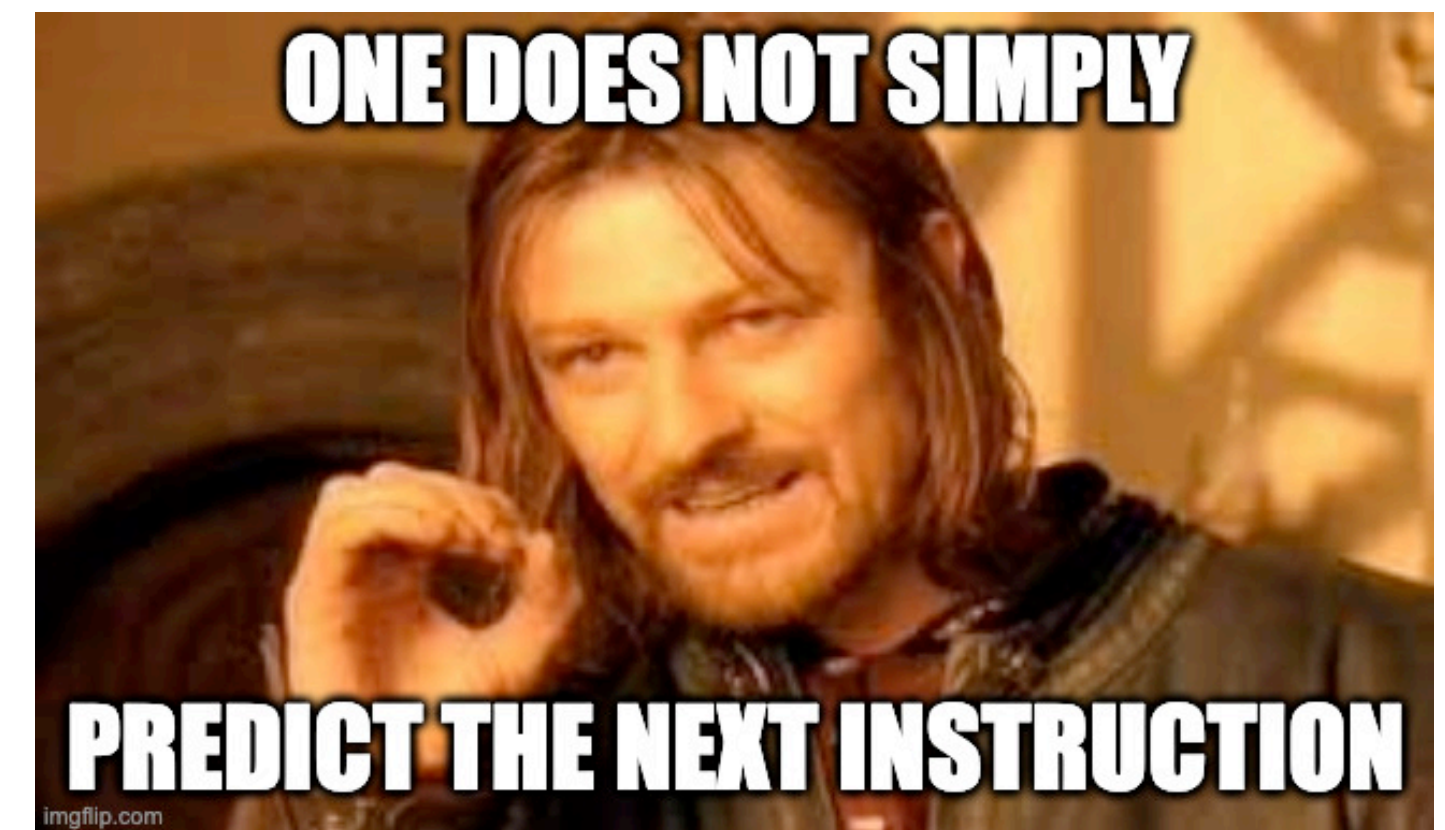
CALL ColoredRectangle >> #width

poly

LOAD aColoredRectangle

LOAD ColoredRectangle

CALL DispatchTable



A Fresh Look at Patching PLCs

- Direct calls: **Good** for prediction
- Invalidation: **Bad** cache behavior
- Not Portable: **Messy** maintenance



A Fresh Look at Patching PLCs

- Direct calls: **Good** for prediction
- Invalidation: **Bad** cache behavior
- Not Portable: **Messy** maintenance



Can we do better?

Patching PLCs are per *send site*

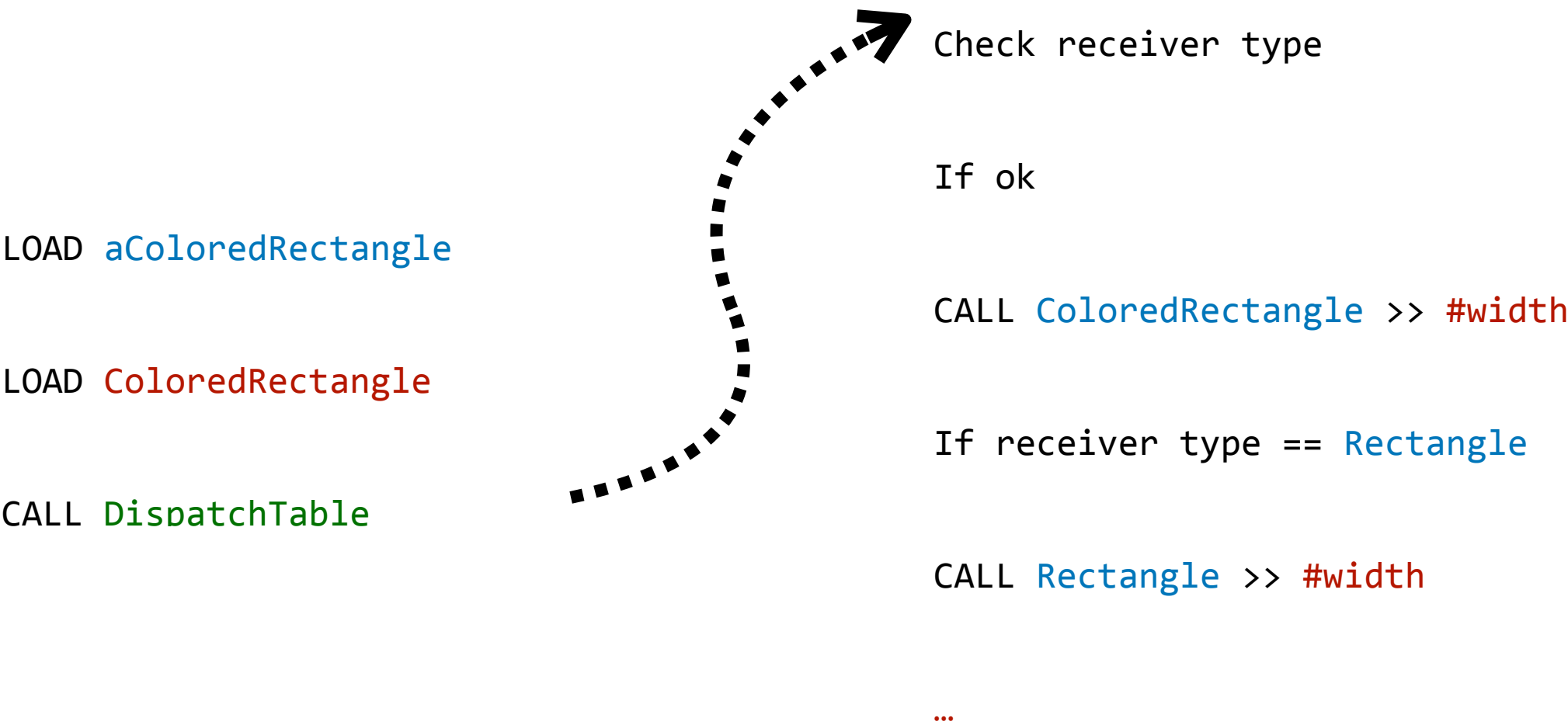
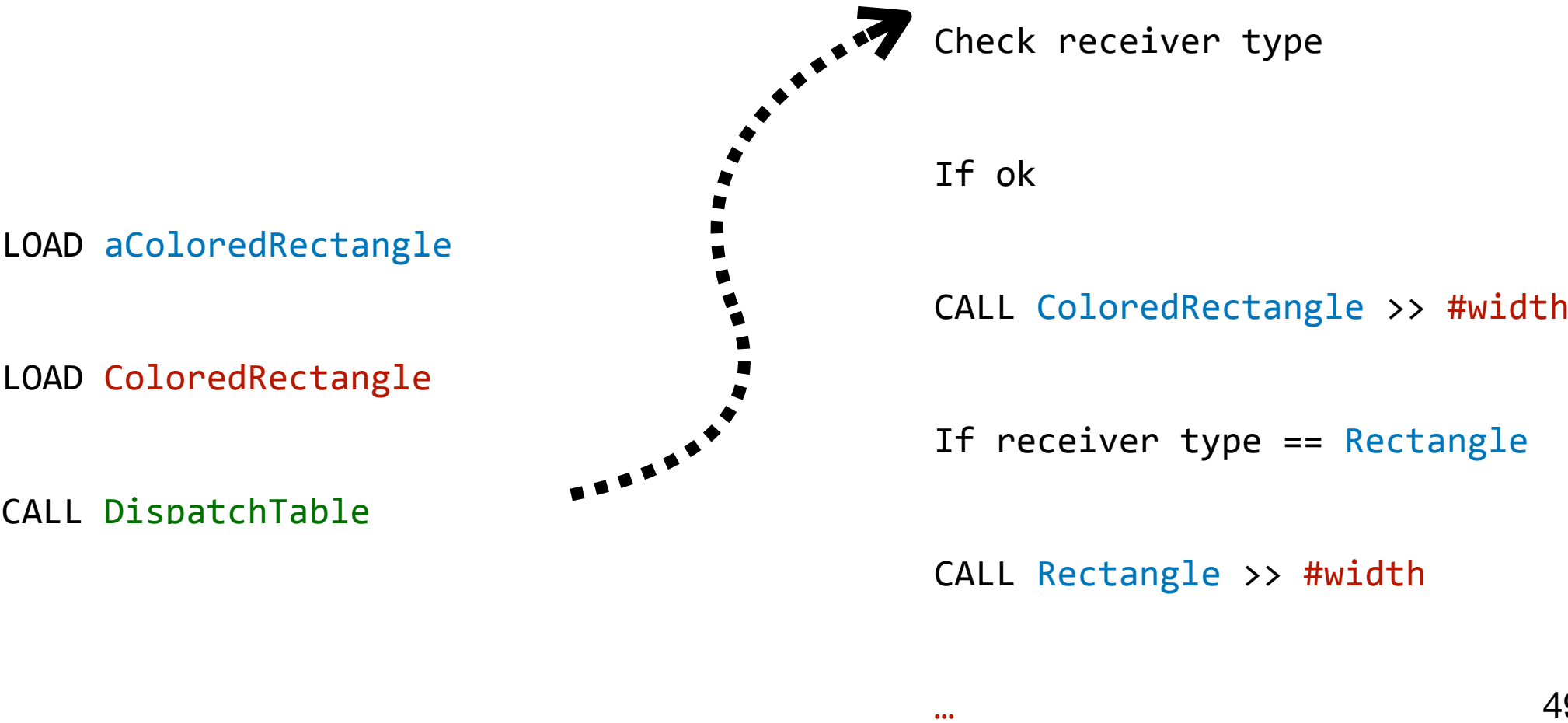
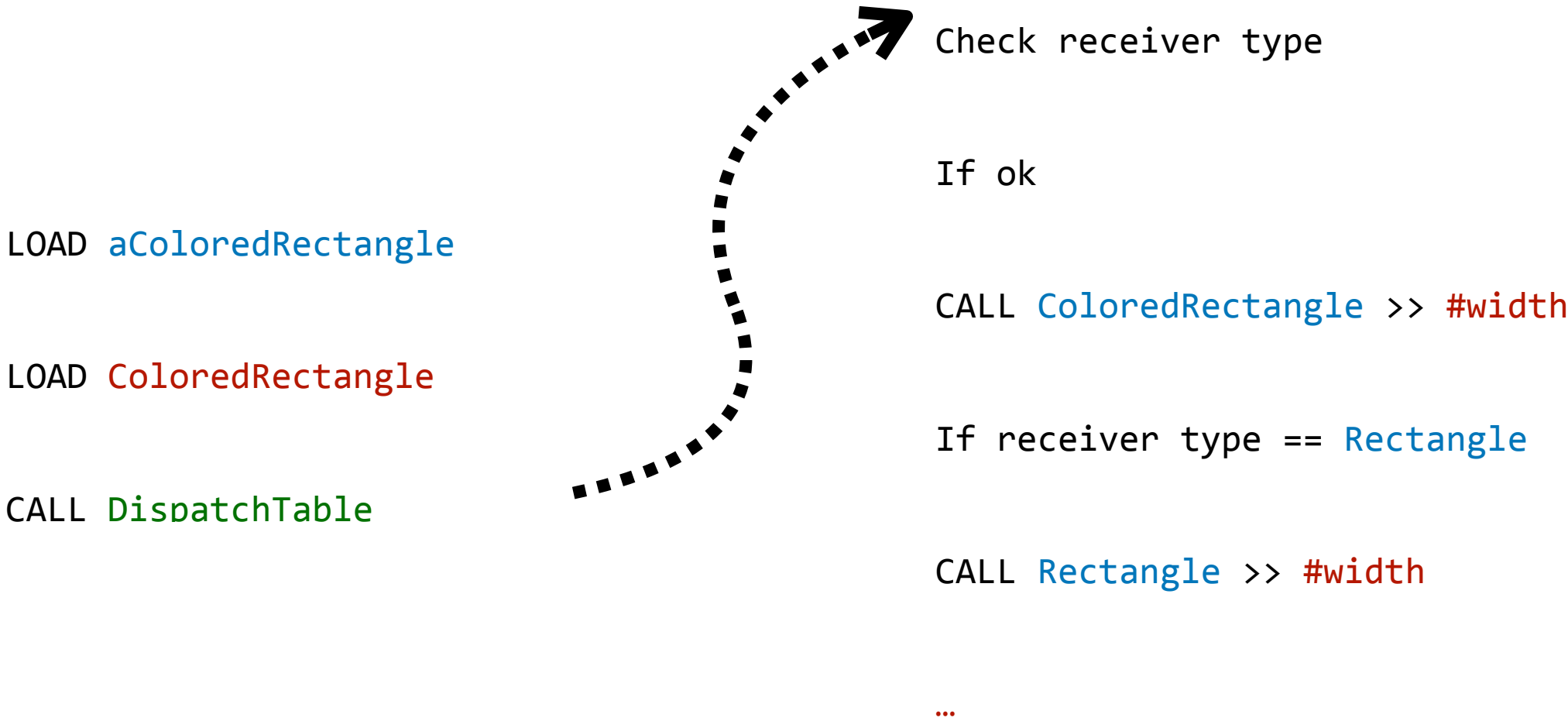
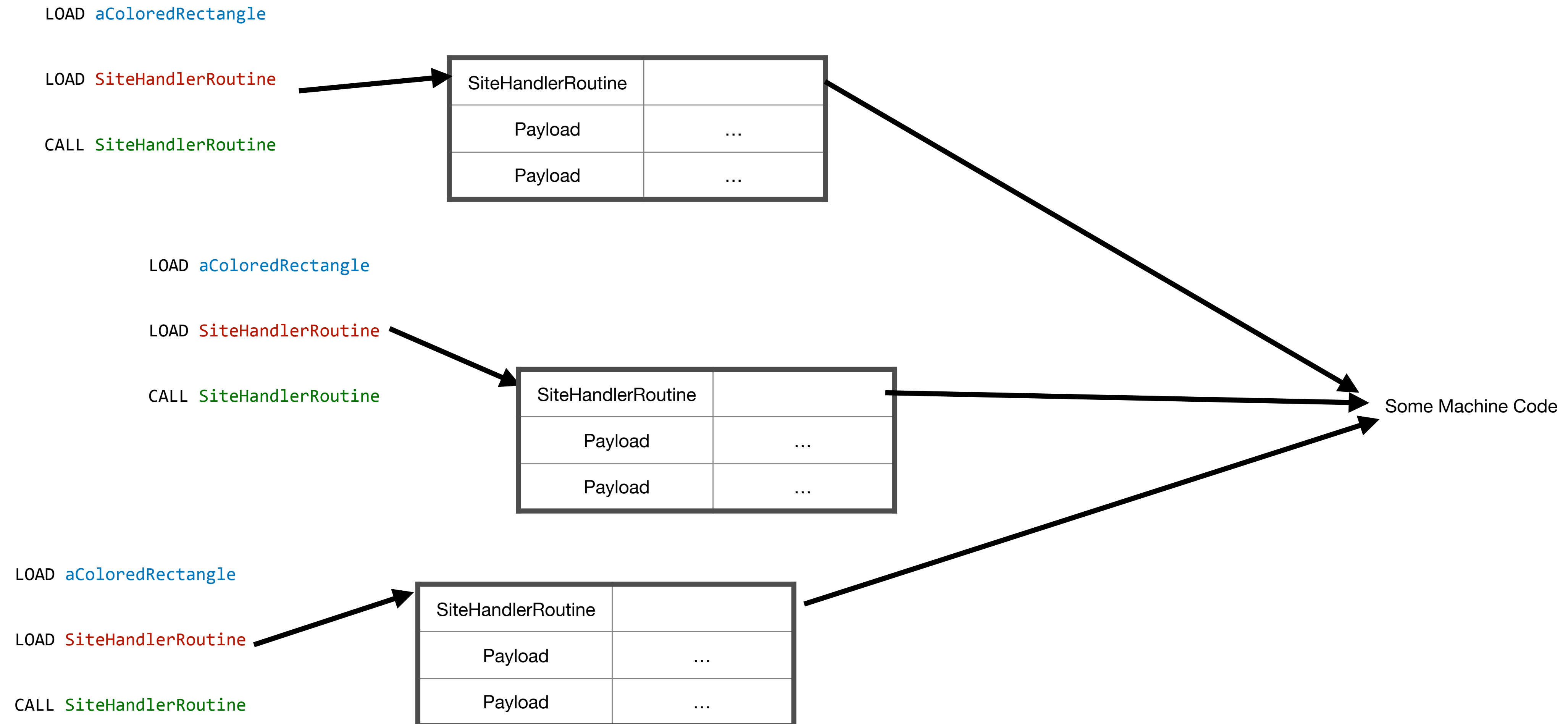
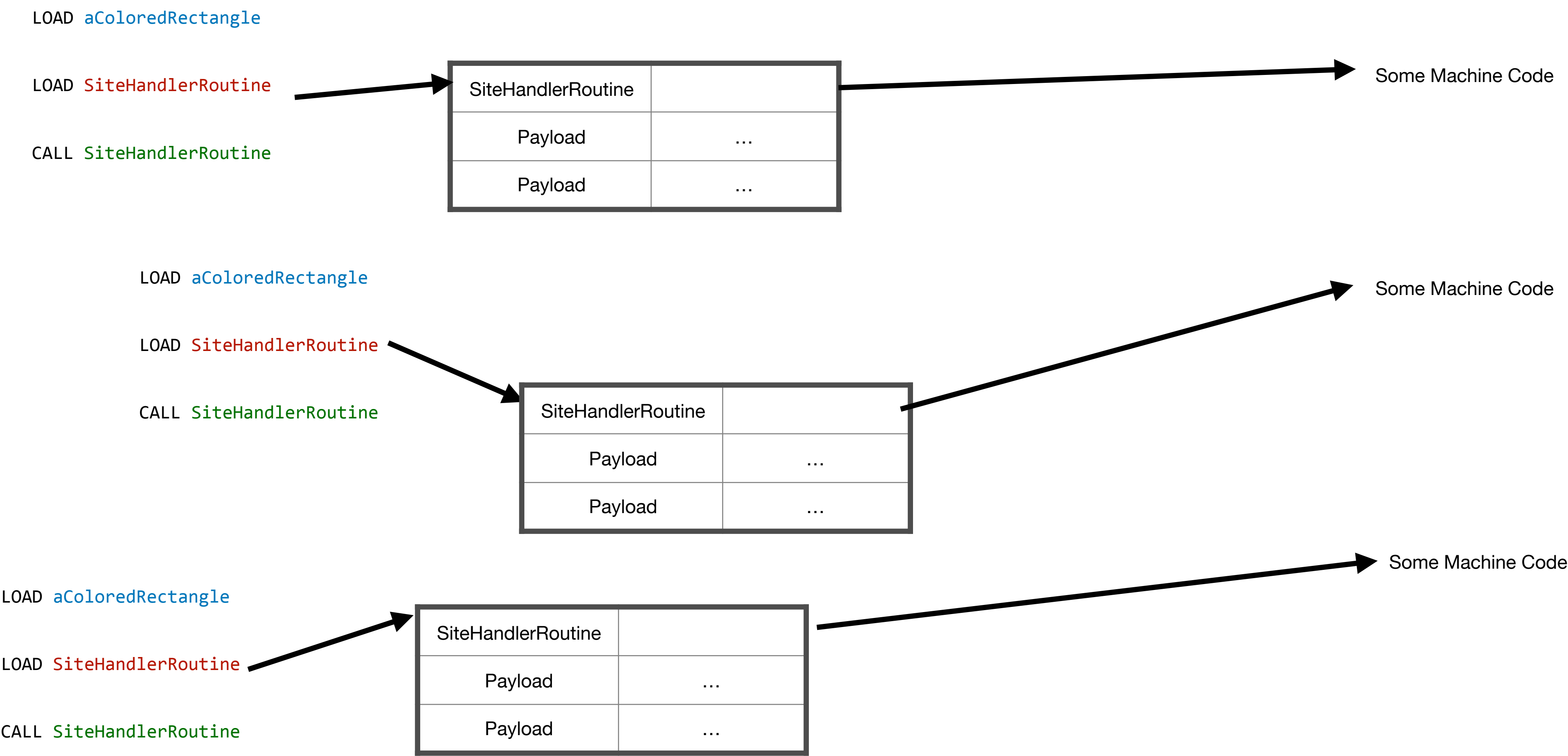


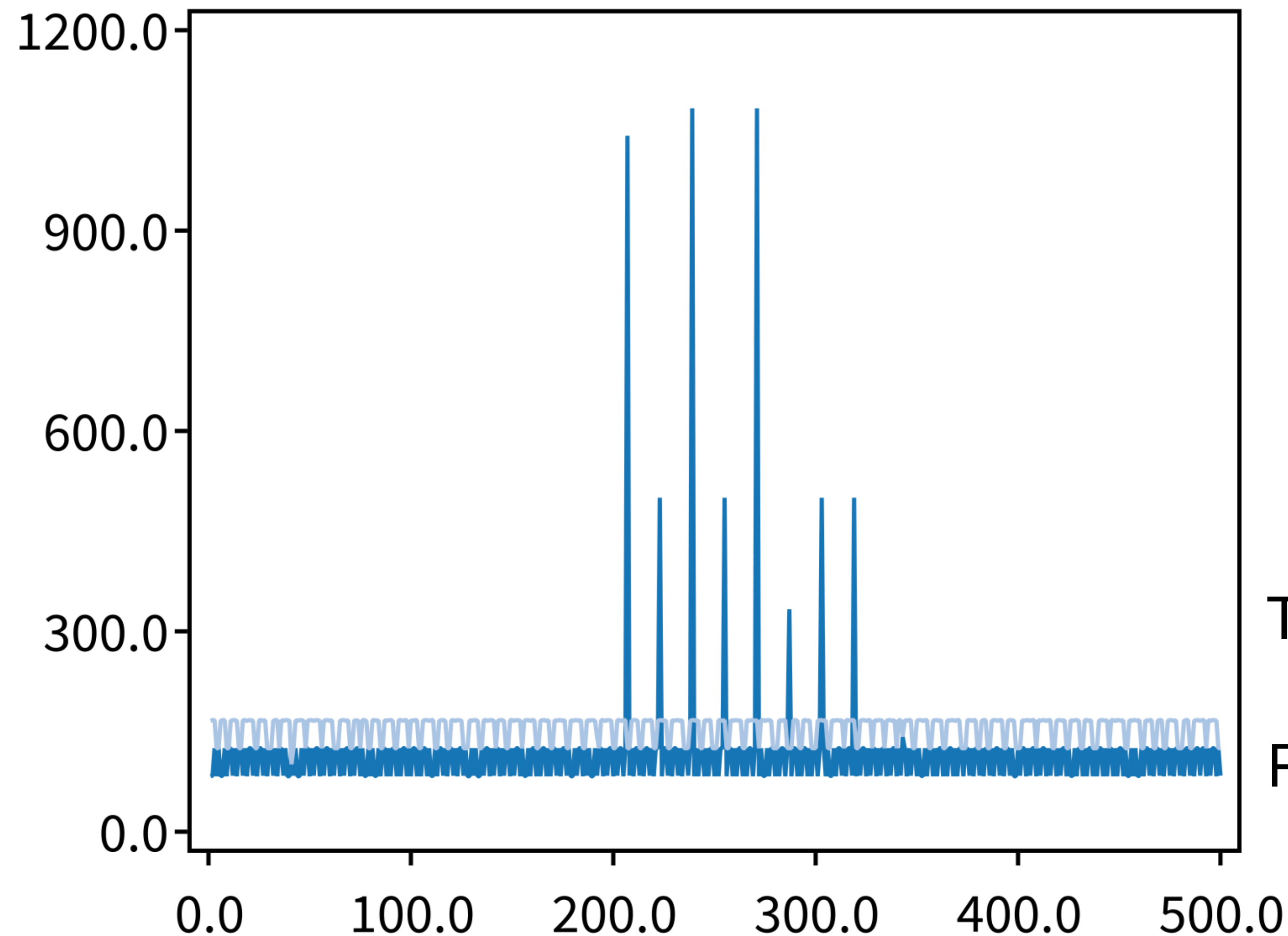
Table PICs *share* the routines!



Let's multiply the routines!



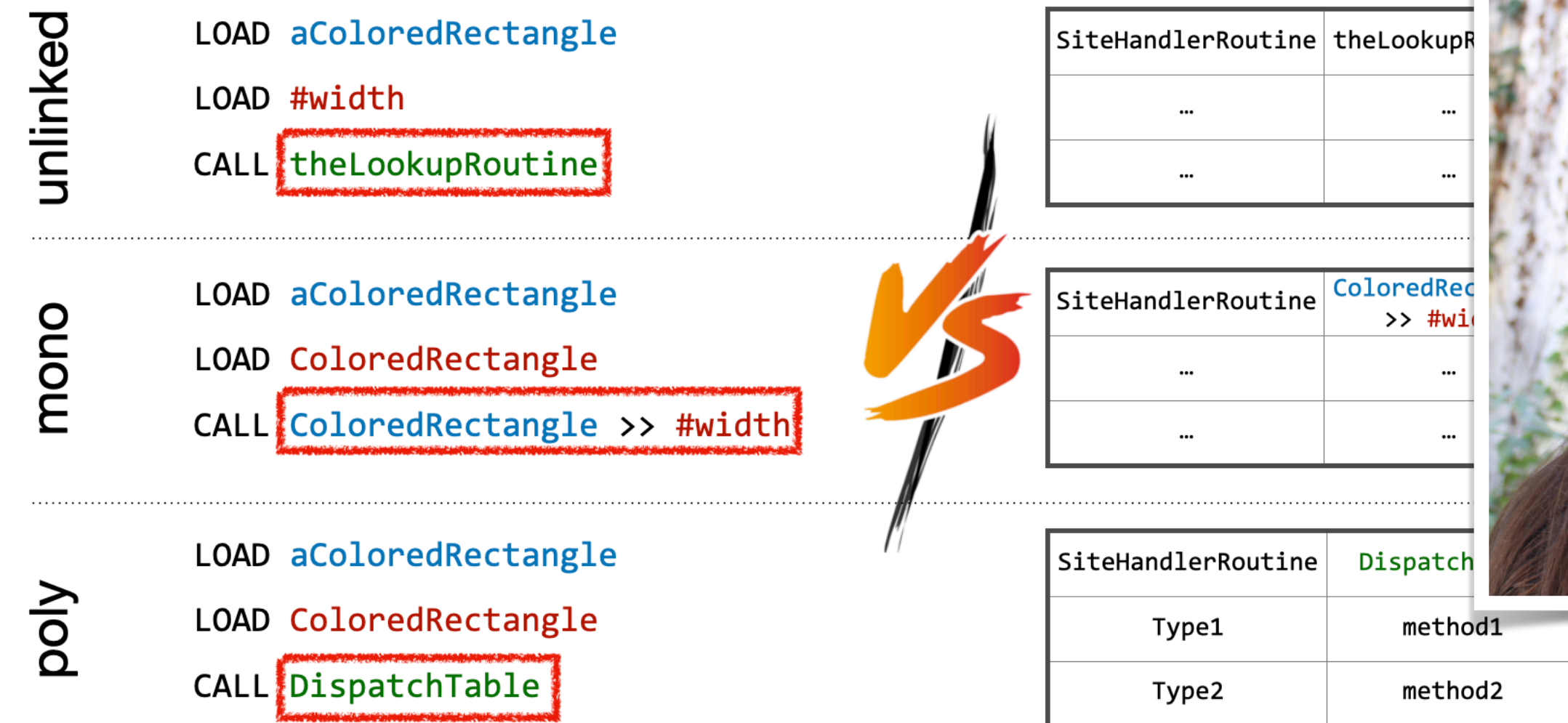
250 stubs, allocated round robin



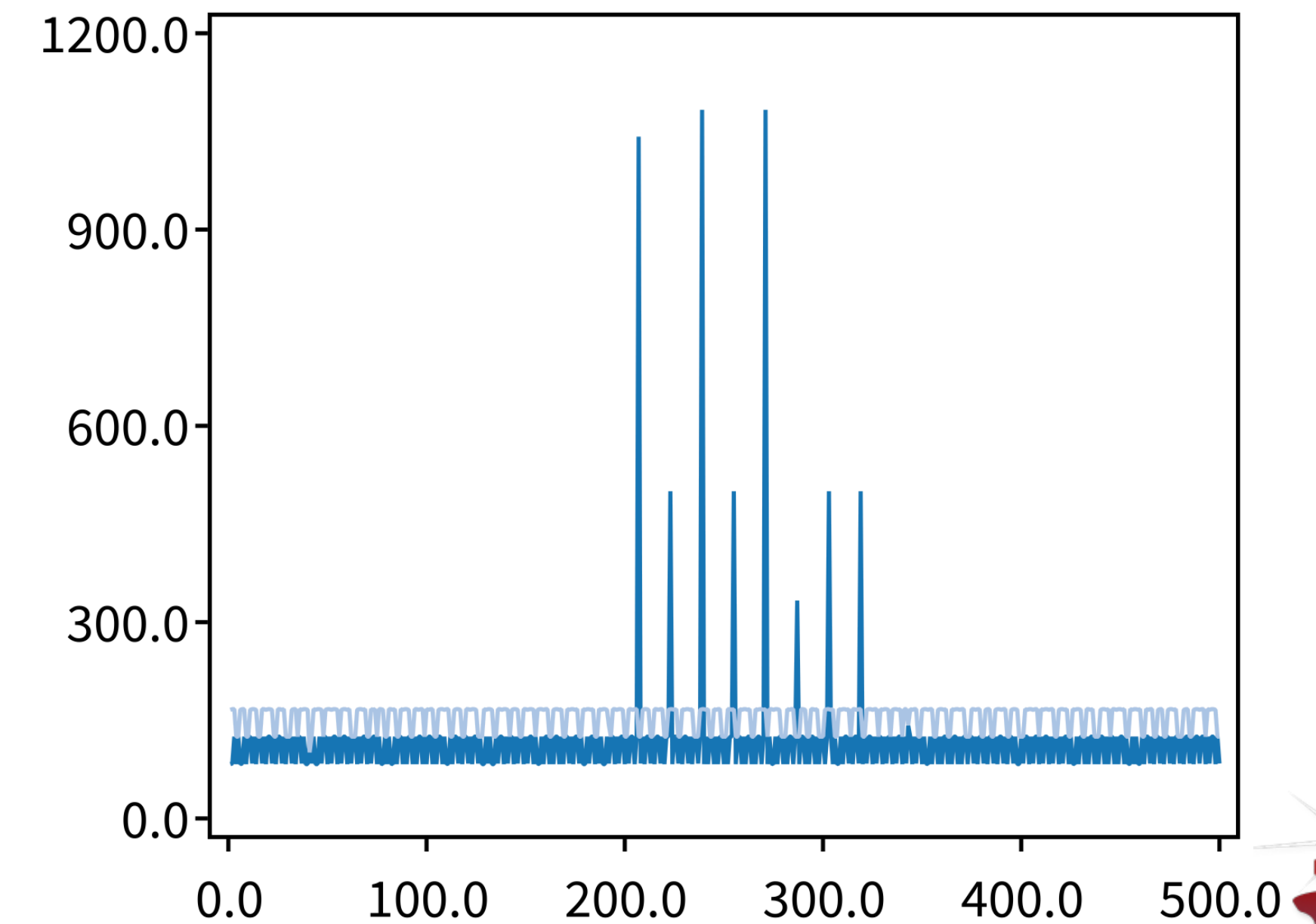
Tables: 154.852 +- 18 **(1.33x)**

Patching: 116.284 +- 85

Takeaways



- **Actively** making the VM **easier to maintain**
- A **simpler** implementation seems **feasible**, but
 - We still need to *help the CPU*
 - **fine tuning** still required

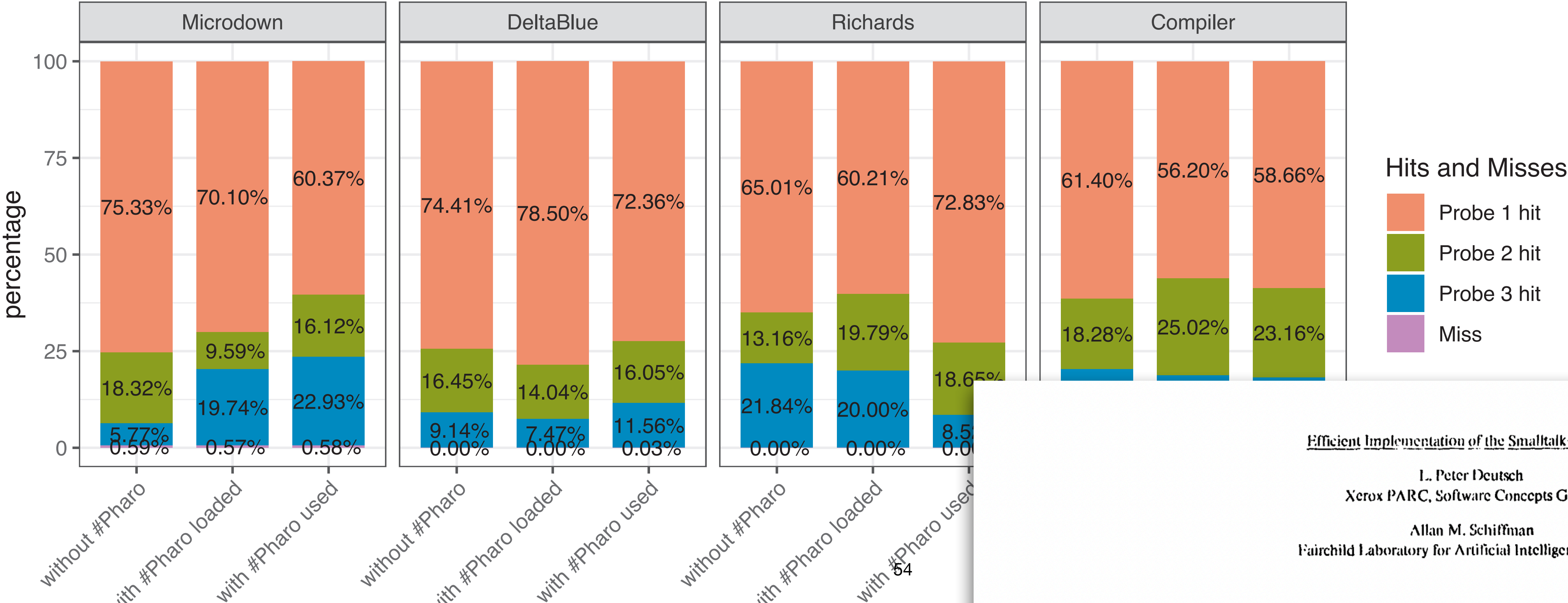


Global Lookup Caches

- Hash table, LRU eviction

```
Interpreter >> lookup: selector in: receiver
method := self lookupInCache: selector in: receiver.
method ifNotNil: [ ^ method ]
```

“SLOW LOOKUP”
...



Efficient Implementation of the Smalltalk-80 System
L. Peter Deutsch
Xerox PARC, Software Concepts Group
Allan M. Schiffman
Fairchild Laboratory for Artificial Intelligence Research